

S. M. GEE & MIKE JAMES

THE ZX81

HOW TO USE AND PROGRAM



S. M. GEE AND MIKE JAMES

The ZX81 How to Use and Program



PANTHER
Granada Publishing

Panther Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

Published by Panther Books 1983

A Granada Paperback Original

Copyright © S. M. Gee and Mike James 1983

British Library Cataloguing in Publication Data

Gee, S. M.

The ZX81: how to use and program

I. Sinclair ZX81 (Computer)

I. Title II. James, M.

001.64'04 QA76.8.S625

ISBN 0-586-06105-3

Printed and bound in Great Britain by
Cox & Wyman Ltd, Reading

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publishers.

This book is sold subject to the conditions that it shall not, by way of trade or otherwise, be lent, re-sold, hired out or otherwise circulated without the publisher's prior consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser.

Contents

<i>Preface</i>	iv
1 Before You Switch On	1
2 Getting To Know Your ZX81	7
3 First Steps – Variables, PRINT, LET and INPUT	25
4 Looping And Choice – The Flow Of Control	40
5 Handling Text And Numbers	58
6 Functions And Subroutines	70
7 Introducing Graphics	84
8 Using Graphics	97
9 Logic And Other Topics	110
<i>Index</i>	122

Preface

The ZX81 is an ideal machine on which to learn BASIC. It may lack some frills – such as colour and sound – but in other respects it is as complete and sophisticated a computer as many that occupy much larger cases! Being small and inexpensive are two of the ZX81's plus points. From the point of view of the novice programmer it has at least two other advantages. First, it uses a well-written dialect of BASIC – which is known as ZX BASIC and is used by all of the machines that have been developed by Sinclair Research. Second, its system of keywords not only makes it easier to enter BASIC programs but, together with its method of checking lines for correct syntax prior to their being accepted by the computer, also makes it more difficult to make mistakes.

All-in-all the ZX81 seems a good machine to choose if you want to find out about BASIC programming without committing yourself to a large initial financial outlay. On the other hand – *unless* you then actually go to the trouble of learning some BASIC programming – you may feel that the ZX81 is not a very exciting computer and you may decide that ZX BASIC is either limited, or slow or even dull and boring. This is very, very far from the truth! What is true, however, is that, unlike some of its more flashy counterparts, the ZX81 is a *demanding* computer – you won't get a great deal out of it unless you put some effort into it.

As far as the ZX81 is concerned, effort need not seem like hard work. Computer programming can be thought of as a stimulating pastime rather akin to playing games of logic like chess or crossword puzzles. ZX BASIC does not have any nasty traps for the unwary and by learning it step-by-step you can quickly achieve a good level of competence.

The ZX81 presents one extra challenge – it is possible to fit

programs of quite remarkable complexity into its original 1K of memory. On the other hand, there still comes the point where this 1K is insufficient for programs that produce interesting effects or produce sophisticated results. For this reason, if you want to follow all the programming examples in this book you'll eventually need to obtain a 16K RAM pack. This transforms the ZX81 into a really useful tool and certainly allows you to develop your programming skills extensively.

As already mentioned, this book adopts a step-by-step approach to programming. The very first chapter looks at computers and programming in general. Next, the ZX81 is introduced and we become familiar with how to set it up and how to use its keyboard. The idea of a variable and three fundamental commands for handling them are thoroughly explored in Chapter 3, and Chapter 4 discusses perhaps the most fundamental idea in programming – the flow of control. In Chapter 5 we broaden our outlook beyond the consideration of numbers and explore the way in which the ZX81 can handle text also. Chapter 6 explains the ZX81's functions and discusses the use of subroutines. The following two chapters are devoted to graphics – both low resolution and high resolution graphics are introduced in Chapter 7 and then both topics are developed in Chapter 8 which presents a useful graph-plotting program and tackles the topic of animated graphics. The final chapter looks at ways of writing better programs and gives hints about how to find the bugs in them but it starts by considering how the ZX81 copes with logic. This may sound like a very advanced subject but in fact it is a very central one to BASIC programming and is one of the features that make a computer into something more than a super calculator.

You will find you get more out of your ZX81 if you make some effort to get to grips with it. In the same sort of way, you'll learn more from this book if you try things out for yourself. There are lots of programming examples to demonstrate effects and for you to try out techniques but, in many instances, they are not the only way of achieving the desired results – and possibly not even the best way! Experiment with the programs, and do try writing your own routines – to embellish, extend and eventually replace the ones provided. Remember, programming is as personal a skill as writing, and developing your own individual style is no bad move.

Chapter 1

Before You Switch On

The Sinclair ZX81 is one of the world's most popular computers and with good reason. For a start, it is inexpensive and so brings the opportunity to join in the fun and excitement of the computer age to hundreds of thousands of people. But the ZX81 is not to be dismissed as a cheap and second-rate computer. Potentially it has power to equal machines that have a price tag even ten times greater. True, when you buy it, the ZX81 lacks some of the frills of the more expensive machines. It has a touch-sensitive keypad rather than a conventional keyboard – something you can change if you wish by buying accessories. There's no sound – unless you buy a sound board as an extra – and there's no colour, though even colour boards are available as add-ons. The one add-on that is essential if you are going to stay the course with this book is the 16K RAM pack. This does not mean, however, that you have to rush out and buy one straight away. The 1K machine is really quite powerful enough for learning BASIC and while working through the early chapters of this book you won't even need to plug your 16K pack on – indeed, it's a good idea to leave it off if the connection between your ZX81 and its RAM pack is at all unreliable. It's only when we come to the subject of advanced graphics that our programs will need the extra breathing space of a full 16K. At this point, although the 1K and 2K machines are actually an interesting challenge – it's quite an art to squeeze your programs into so little space! – things quickly start to get frustrating simply because you are always having to worry about running out of memory. Once you have your 16K on board you have a respectably sized computer. After all, only twenty years ago a computer with a similar memory capacity would have filled the whole of a family house! However, the fact that modern microtechnology has enabled so much to be crammed into so small a space is a fact that the ZX81

user can cheerfully ignore, for at the same time as computer design is becoming more sophisticated, computer use is becoming accessible to everyone who wants to join in.

What is a computer?

This question is one that can be answered at many levels. Whole chapters, even whole books, can be devoted to the subject. In order to use your ZX81, however, you really do not even need to ask this question. After all, we all watch TV but few of us ever question what exactly a television is. If you want to know every detail of the way the ZX81 works then there is no choice but to learn about electronics. However, it is not difficult to gain an understanding of what a computer does and roughly how it does it without knowing anything about electronics or the ever-present *chip*! The point is that a computer is a device that would exist even if electronics had never been invented. Indeed, the first computers were built using cogs and gears and it took one hundred years before a valve (an early electronic component) found its way into such a machine. Although in practical terms the computer seems to be a product of microprocessor technology, the idea that lies behind a computer doesn't depend on the materials that you choose to build it from.

Every computer is composed of a number of parts that each perform a well-identified function. Any computer has to have some way of communicating with the outside world. In the case of the ZX81 this need is met by a keyboard, which you type on, and the TV screen, which the ZX81 can use to show you what you have typed and anything else it needs to tell you. The keyboard is an example of an *input* device and the TV screen is an *output* device. These are not the only input/output or I/O devices that can be used with a computer. You can buy a small printer that can be used with the ZX81, for example, and you can direct the ZX81 to produce its output on this printer instead of, or as well as, the screen.

A machine that could only receive information and pass it on unchanged wouldn't really be worth calling a computer. Rather it might be classed as a telephone or a telex! Inside every computer there has to be some mechanism that can change or *process* information before it is printed out. This mechanism usually takes the form (these days, at least) of complex electronics hidden inside the computer. What we are talking about is often referred to as the

Central Processing Unit or *CPU* but it also has a traditional English name that betrays the fact that computers were once made of cogs and gears – the *mill*.

In the ZX81 the CPU is contained in a single chip known as a *Z80* and this is, of course, the origin of the Z in the ZX81's name. What exactly the Z80 does isn't of too much importance from the point of view of programming in BASIC and the way that it does it certainly isn't! In general, however, what the Z80 does is to perform arithmetic and other operations on information input from the keyboard and stored within the machine. What operations it does are controlled by a list of instructions called a *program*. This aspect of a computer is so important that you could almost say that a computer *is* a machine that will obey a list of instructions – but any sort of definition of a machine as complicated as a computer is dangerous! What sort of instructions the ZX81 can obey will occupy the rest of this book so, for the moment, the subject will be set aside.

If a computer is going to obey a list of instructions concerning what to do with various pieces of information, it must obviously have somewhere to store not only the information but also the list of instructions. This part of a computer is known as *memory* but the slightly less general term RAM (standing for *Random Access Memory*) is almost universally used instead. You can think of RAM as a sort of note-pad where the CPU can record its list of instructions and any data that it needs. Obviously every memory has a limited capacity and this is an important measure of how powerful a computer is. The larger the memory the larger the list of instructions that can be stored. The most convenient unit of measurement to apply to computer memory is the *byte*. Roughly speaking a memory that can store one byte can store one *character*. (Here the term 'character' means a letter, a digit or any punctuation that you might find in a normal text – such as this book!) So a 400 byte memory could store enough characters to hold about a quarter of a page of this book. The only trouble with this convenient unit of measurement is that it is a little too small. Computers normally have memories that can store thousands of characters and so it makes good sense to think in terms of thousands of characters. The unit used for this is the *kilobyte*, which is often shortened to *Kbyte* or even just *K*. For various reasons, however, 1 Kbyte isn't 1000 bytes as its name suggests, but 1024 bytes. (You may notice that this strange number is the nearest power of *two* to 1000 and, as you might already know, computers

work in binary which is based on *two* states). The 1K ZX81 therefore can only store just over a thousand characters, which sounds a lot to begin with but quickly begins to seem limited once you realise that a single variable actually takes up eight characters. However, once you've clipped on the 16K RAM pack, your ZX81 can store roughly 16000 characters which is enough for a wide range of interesting applications. Early computers that were used by the military to calculate missile trajectories, etc. often had less than 16K!

This combination of I/O devices, CPU and memory is all that there is to a computer. The I/O communicates with the outside world, the CPU calculates and generally processes information, and the memory holds the list of instructions that the machine obeys and the data that the CPU acts on. In practice, there is one addition that we must make to this list. When you switch your ZX81 off it *forgets* everything stored in its memory. To keep information stored accurately, most computer memory needs a constant supply of electricity. If you switch off the supply the information is lost. This sort of memory is often known as *volatile* memory. This loss of memory is something of a problem because it implies that we have to type in the list of instructions every time that the ZX81 has been switched off. To overcome this difficulty most computers have a second form of memory that is *non-volatile*. In the case of the ZX81 this takes the form of a standard cassette tape recorder that can be used to *save* programs and data in a form that exists even when the power has been switched off. A second advantage of this type of memory is that it is removable. You can record a program on a cassette and then take it out of the recorder (and even send it to someone else). The ZX81 is then ready for you to start on a new program or to go back to an old one, which you can do by *loading* it from an earlier recorded tape.

Programs and programming

As mentioned earlier, a computer obeys a list of instructions stored in its memory. This list of instructions is known as a program and writing such lists of instructions is known as programming. It is often thought that programming is an activity that started with the modern digital computers but people have been writing lists of instructions for other people to obey since writing was first invented. In this sense, programming is nothing new and can be seen in the form of recipes and knitting patterns in almost every home. Perhaps one of the best

examples of traditional programming is written music. You can think of sheet music as being a program that will instruct a musician to play a specific tune. In fact, written music is very like a computer program in that it relies on using a special language that is much more precise than ordinary language. Just one note out of place and you have a different tune! A computer program is written using a special and equally precise language. In the case of the ZX81 this language is BASIC, the most popular programming language in the world. Just as with written music, slight changes in a BASIC program can alter its meaning completely, so it is important to realise as you learn BASIC that you must pay attention to the fine details right from the very beginning. Unlike learning English, where you can first learn words and sentences and then add punctuation, you have to take notice of every comma in a line of BASIC for it to make any sense at all!

If all this talk of strict rules is worrying you it is worth saying that the rules are usually very simple and very regular. Unlike English there are rarely any exceptions to spelling and punctuation rules in BASIC! In addition, there are some powerful underlying ideas behind BASIC. Once you have recognised these they make it easy to understand why the rules are there at all. As you progress through this book there are therefore two types of thing that you will learn – the fine detail concerning the exact form of each BASIC statement, and the general features that all programming languages share. The fine detail is important to actually getting a program working but understanding the general features make the act of programming a sensible occupation.

The history of the ZX81

Before moving onto a discussion of the ZX81 it might be of interest to take a brief look at its family tree. In 1980, Sinclair Research launched a small plastic-cased computer, the ZX80, that brought computing within the reach of nearly everyone. The trouble was that the ZX80 was very limited. It was a revolution but in many senses it was just a little before its time. It could be used to run small programs written in BASIC but the sort of arithmetic that it could do was restricted to whole numbers. It could display information on a TV screen but only while it wasn't processing data. If it was doing anything at all useful the TV screen flickered disturbingly. This meant that many people who bought the ZX80 for various reasons were

disappointed to discover that it just couldn't live up to their expectations of a computer.

In 1981 Sinclair launched the successor to the ZX80, namely the ZX81, the subject of this book. It hit the USA in 1982 where it also has another identity, the Timex 1000. Meanwhile, in 1982 Sinclair brought the Spectrum onto the British market. However, the ZX81 did not suffer the fate of its predecessor – the ZX80 had virtually been abandoned overnight – because it had, even in the brief space of its existence, built up a reputation and a following. One of the chief reasons for this was its language. The ZX81 remained notable because it introduced Sinclair or ZX BASIC. ZX BASIC is a fully developed version of BASIC that has many advantages over the BASIC on other machines. To make up for the small memory size, Sinclair produced a 16K add-on RAM pack and a small low cost printer was developed to extend the machine's range of use.

The ZX81 is not going to lose its place as a popular and inexpensive machine. ZX BASIC has also taken its place as an important and popular programming language. Learning this BASIC is, therefore, likely to stand you in good stead for now and for the future.

Chapter 2

Getting To Know Your ZX81

There are two problems in using a computer. The first is simply getting it set up and getting used to its idiosyncracies. The second is writing working programs. This chapter deals with both these problems so that we can get them out of the way before we get down to the main task of learning BASIC.

A new acquaintance

Getting to know a computer is a problem that exists even if you're an expert. For although there is a lot in common between different computers, there are always enough little differences to mean that there has to be a period of adjustment when moving from one machine to the next. For example, nearly every computer uses a standard typewriter (or QWERTY) keyboard but most place extra but very important keys in slightly different places and this can make even the most expert look silly at first! Now, if you're an expert, then you know that this early phase soon passes but if you're a beginner you might panic and think that computing was always going to be this tricky! The trouble is that not being 'at home' with your computer can make easy programming ideas seem difficult.

There is no way to avoid this early barrier to programming because being on friendly terms with your computer is simply a matter of time and a matter of using it. You'll come through this rather frustrating period more easily if you bear in mind the following advice:

- Separate in your mind any difficulty that you encounter in using your ZX81 from any difficulties that you have with programming.
- Don't immediately assume that any strange behaviour on the part of your ZX81 is its fault – at first the chances are that the mistake is yours!

- Don't immediately assume that any unexpected behaviour of a program means that your ZX81 is illogical – computers are ruthlessly logical. Try not to confuse typing errors with programming errors.

To try to help you identify this initial difficulty and to help you overcome it, this chapter includes two short programs that you should try to get running on your ZX81 before moving on to the rest of the book. They are presented as complete and working programs for you to use to find out about the *non-programming* problems of using the ZX81. At this stage you are not expected to be able to understand how they work and you might like to return to them as you read through later chapters to see how you're progressing.

Setting up the ZX81

Once you have done it a few times you'll find that it's really very easy to get your ZX81 working. In order to get started initially there are just two leads to be connected – one to the TV set and the other to the power supply. Connecting the power supply is very simple. All you have to do is to plug the small black box marked 'ZX POWER SUPPLY' into the mains and then insert the small jack plug into the socket marked '9V DC' on the side of the ZX81. Connecting the mains to your ZX81 is the only thing that is required to make it start working. However, if you want to see what it is doing you will have to connect it to a working UHF TV set! This operation is a little more difficult than connecting the mains to the ZX81 because it involves a piece of equipment that will vary from house to house – the TV set. When connecting the ZX81 to the TV set, it will help to think of it as being an extra channel. In other words, the ZX81 is not only a computer, it is a small television station! When your TV set was first delivered it had to be tuned in to receive the channels that are used in your area. In the same way, when you first use your ZX81 you have to tune the TV set to receive it. If your TV set has push-button tuning then you will have to decide which button is going to be the 'ZX81 channel' and find out from the instruction manual that came with the set how to tune it in. If your set has dial tuning, then finding the ZX81 channel is exactly the same as finding any other channel. To make things easier, before you connect the ZX81 to the TV set, either tune the channel button that you are going to use or set the dial to receive BBC 2. (Non-UK readers must consult the tuning information that came with their ZX81 to discover their equivalent of BBC 2.) The

reason for this is that the ZX81 channel is just a little higher than BBC 2 and rather than start the search from anywhere it is easier to start from BBC 2 and then tune the set away from the other BBC and ITV channels toward channel 36 – which is the one the ZX81 uses.

Once you have tuned to BBC2, remove the TV aerial from its socket and replace it by the aerial lead that came with your ZX81. Then plug the other end of this lead into the socket marked 'TV' on the side of the ZX81. Now with both the ZX81 and the TV switched on (you should turn the TV's sound right down to avoid unpleasant noises) start tuning the TV set. Keep going until the letter **K** appears clearly in the bottom left-hand corner of the screen. Just as with any TV channel, if you haven't tuned in exactly the picture quality will be poor, so take care and be patient with the fine tuning until you have a nice sharp image. You may have to adjust the brightness and contrast controls to get the best possible picture but don't do this until you have finished tuning for the sharpest picture or things will get hopelessly confused. If you prefer, you can leave any final adjustments until you have entered the program which provides a 'test card' for your ZX81 given later in this chapter (see Fig. 2.4).

Using the keyboard

After getting your ZX81 going, the next thing to do is to start to learn your way around the keyboard. The first thing to get used to is the fact that it is a touch-sensitive keyboard rather than a conventional one with key switches. If you find this difficult you can purchase an alternative keyboard from your dealer but personally we find the keyboard perfectly satisfactory as it is.

Perhaps the most off-putting feature of the ZX81 is the number of letters and words that are written on and around the keys of the keyboard. The idea of having each key on a keyboard perform more than one job is not a new one. Most typewriters use a single key to print lower-case (small) and upper-case (capital) letters and no-one seems upset by the idea of selecting between the two by pressing an extra key – the *shift key*. Notice that the shift key doesn't actually print anything if you press it all on its own so it's not the same as the other keys on the keyboard. As it controls what the other keys produce, it is known as a *control key*.

The ZX81 actually has a number of control keys which between them give access to all its characters, commands and features. The

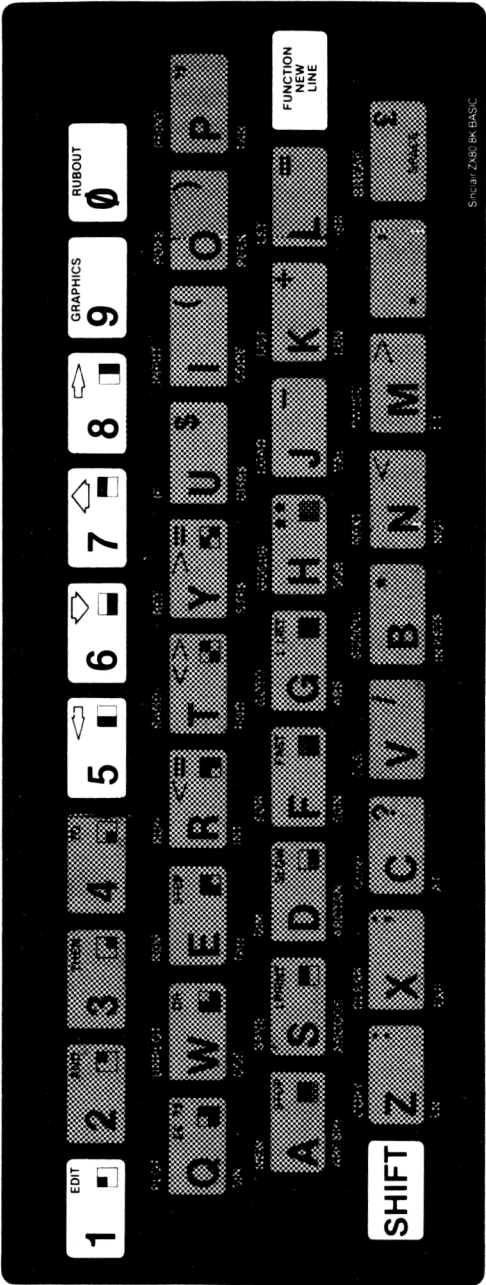













Fig. 2.1. The position of the control keys.

diagram (Fig. 2.1) shows their location on the keyboard. The function of all these control keys will be explained in this chapter, starting with the ones needed most frequently.

When you first switch on your ZX81 and press any of its letter keys a whole word appears. You may find this disconcerting at first but it is actually a very helpful feature which can greatly speed up the task of writing your own programs. The reason for this facility is that every line of BASIC begins with one of a small set of words, the *keywords*. To make life easier, the ZX81 interprets the very first letter key that you press as a keyword. The keyword produced by each key is written in over each of the letter keys. The keyword on the A key is NEW so when the A key is pressed for the first time the word NEW appears. If you press the A key a second time then the letter 'A' is printed on the screen. To let you know what any key is going to produce when you press it, the ZX81 displays a different letter in the flashing square that it constantly displays while you are entering commands. The flashing square is known as the *cursor*. Its position on the screen indicates where the next printing position is and the letter displayed gives information about which set of keyboard characters will be used. At the start of every line the cursor is a flashing  until a key word is entered and then it changes to a flashing . ( stands for keyword and  stands for letter.)

The ZX81 only has upper-case letters and the SHIFT key has a rather different function to that on a typewriter. While the cursor is showing either  or  and you press the shift key with one of the letter keys then the red word or symbol at the top of that letter appears on the screen. Similarly, if you press SHIFT and the 2 or the 3 key on the top row when the cursor is  or  you will see the word AND or THEN respectively.

The words printed in white below each of the letter keys are collectively known as the ZX81's *functions*. In order to gain access to them you have to press the SHIFT key and the NEWLINE key (which you will notice has FUNCTION printed in red on it) simultaneously. Doing this causes the cursor to change to  and if you press any key on the keyboard while the  is flashing the word directly below it will appear on the screen. Notice also that as soon as you have entered a function the cursor changes back to  immediately.

All this may seem rather confusing in the abstract so it is best to consider a single key and try to figure out how to obtain each of the

words and symbols written on and around it (ignoring the black and white square to the left of the R for the time being). It's a good idea to try working this out for yourself before reading the summary given here for the R key (see Fig. 2.2).

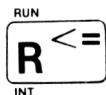







Fig. 2.2.


- At the beginning of each new line the cursor is a flashing . Pressing R will result in RUN being displayed.
- When the cursor is , pressing R will result in R being displayed.
- When the cursor is  or  pressing SHIFT and R will result in <= being displayed.
- When the cursor is  (obtained by pressing SHIFT and NEWLINE together) pressing R will result in INT being displayed.

We now know how four of the symbols or words that surround each key can be produced and this is enough to enter a short program. There is more to using the keyboard but we will return to this later.

Entering a program

As explained in Chapter 1, a program is a list of instructions that a computer can obey. In BASIC this list of instructions is built up by typing in lines of commands, each one beginning with a number – the *line number*. The easiest way to understand this is to try out a short program. Don't worry if you do not understand how the program works – just concentrate on entering it correctly. Before you begin, switch your ZX81 off and then, after waiting a moment, on again. This will ensure that anything you may have typed in while experimenting is cleared out of the machine. Now type the following line, taking care not to make a mistake. (If you do make a mistake then switch off and start again. This is only a temporary way to overcome mistakes. More satisfactory methods will be explained shortly.)

10 LET A=0


Notice that the word LET is a keyword and is produced by pressing L while the cursor shows a flashing . Also notice that the last character is a zero, not a letter O (the zero is on the far right of the top row of keys). The '=' sign is in red on the L key and so to enter this you have to press SHIFT and L.

After you have typed this line you have to press the key marked NEWLINE on the far right of the keyboard. The purpose of this is to tell the ZX81 that you have finished typing the line and that it can try to incorporate it into any program that you may have already typed. The word 'try' is used because, even though you may think that what you have typed is correct, the ZX81 checks it and, if it finds that you have typed nonsense, it will refuse to accept it (but more of this later). When the ZX81 accepts the line it disappears from the bottom of the screen and appears at the top. Now enter the following line:

```
20 PRINT A
```

Once again, the word PRINT is a keyword and is entered by pressing one key, in this case P. Press NEWLINE, and the second line will appear, again in the upper part of the screen, just below the first line. Next, enter the following line:

```
30 LET A=A+1
```

This time, before you press NEWLINE, let's see how you could correct any errors that you might have made. Suppose, for example, that by mistake you had typed B=A+1. You might then be pleased to know that the ZX81 offers you a backspace facility which enables you to change the line you have typed in very easily. If you press the left-arrow key (the 5 on the top row) while you press SHIFT you will see the cursor move over the letters that you have entered. If you press the RUBOUT key (the  on the top row) while pressing the SHIFT the letter or even a whole keyword to the left of the cursor will vanish (i.e. will be deleted). You can insert new keywords or letters simply by typing them in. You can move the cursor to the right by using the right-arrow key (the 8 on the top row) and any new characters that you type will be inserted to the immediate left of the cursor. (This is all a lot easier to see happening than it is to explain so don't be afraid to experiment – you can't hurt your ZX81.) Finally, when you have finished entering the line, press NEWLINE. Next, type

```
40 GOTO 20
```

You should now have the following program in the top part of the screen:






```
10 LET A=0
20 PRINT A
30 LET A=A+1
40 GOTO 20
```

This is a list of instructions that you can make the ZX81 obey by entering the keyword RUN and then pressing NEWLINE. But before trying this it is worth noticing that there are two ways that the ZX81 can obey commands. If you type a command without a line number and press NEWLINE then the ZX81 will obey the command at once – this is called *immediate mode*. However, if you precede the command with a line number and press NEWLINE then the command is added to whatever program already exists and is waiting to be obeyed at some later time – this is called *deferred mode*. When you type RUN there is no line number so the ZX81 obeys the command immediately.

When you do type RUN you will find that the numbers from 0 to 21 are printed on the screen. The message that the ZX81 prints at the bottom of the screen, '5/20', informs us that the screen has been filled before the program has finished. If you type CONT (by pressing the 'C' key) the program will CONTinue. Try this and then try breaking into the program while it is still running by pressing the BREAK key – at the bottom right of the keyboard.

Once you have stopped the program running, try issuing another command in immediate mode – type LIST (the key marked K) and press NEWLINE. This will cause the program you have typed in to be displayed on the screen.

More about the keyboard

So far we have considered what happens when keys are pressed when the cursor is flashing ,  or . The cursor has yet a fourth possibility. It can become a flashing  and when that appears on the screen you have access to the ZX81's graphics characters. To obtain the  cursor, press the SHIFT key and the 9 key simultaneously. Now if you press any of the letters keys you will see them appear in *reverse video* mode – that is, white on black rather than the normal black on white. To display one of the graphics symbols on either the number keys or the letter keys press it together with the SHIFT key.






Once you have entered the graphics mode the  cursor will remain until you press the SHIFT key together with 9 a second time. Returning to our examination of the R key we can now add two final possibilities to our previous list (see Fig. 2.3).



Fig. 2.3.

- When the cursor is  (obtained by pressing SHIFT and 9 together) pressing R will result in R being displayed in reverse video (i.e. a black R in a white box).
- When the cursor is  (obtained by pressing SHIFT and 9 together) pressing R with the SHIFT key held down will result in a corner-shaped black character being displayed.

By now you should be able to produce any of the characters, words or symbols on or around the letter keys. That just leaves us the number keys to consider. When the cursor is either  or  pressing one of the number keys will produce the appropriate number. Pressing SHIFT and the number 2 will produce AND, while SHIFT and 3 display THEN. We have already met SHIFT and Ø, which is RUBOUT – the delete facility – and SHIFT and 9, which switches you to graphics mode. Once in graphics mode, pressing the number keys displays them in inverse graphics while pressing the numbers 1 to 8 together with the SHIFT key produces the symbols to the left of the number.

Editing your programs

Pressing the SHIFT key and the 1 key allows you to *edit* lines of programs that you have previously typed. The editing facility relies also on the four cursor control keys, that are obtained by pressing the SHIFT key together with the 5, 6, 7 and 8 keys. The arrows above each of these keys indicate the direction in which the cursor will move when you use them. We have already discussed the use of the 5 (left) and 8 (right) keys to move the cursor's position within the current line. The 6 (down) and 7 (up) keys allow you to move the cursor's position up and down the screen. You will notice that every time you add a new line to your program the cursor is displayed between its line

number and its content. Using the up and down cursor control keys you can, however, make another line of your program the centre of attention. Pressing SHIFT and I has the effect of displaying the line where the cursor is positioned in the text input area at the bottom of the screen so that you can change it. Once you have made any desired change, press NEWLINE and the corrected line will resume its position within the existing program. Once again, this is an idea that seems difficult in the abstract but is quite straightforward once you have tried it for yourself. The next demonstration program will give you an opportunity to experiment with it.

There is one other point to make about the four arrow keys. They are often made use of in games programs. When they are used in this way they usually work without the SHIFT key so that pressing the 5 key on its own causes the cursor to move to the left and so on. The ability to make the arrow keys work in this way without the SHIFT key has to be specially programmed to be different from their normal operation.

Using a tape recorder

Our second example program is considerably longer than the first. As it is likely that you will be loath to simply switch the machine off and so lose the program altogether once you have finished entering and running it, now is the time to learn how to use a cassette recorder to save and load programs. The ZX81 can use almost any standard tape recorder to store programs, but it is true that the better the tape recorder the more reliable the result. So if you are thinking about buying a cassette recorder for use with your new ZX81, invest at least £20 or so and try to buy a model that uses miniature jack sockets for earphone and microphone connections. If the tape recorder that you plan to use doesn't use miniature jack sockets then you will have to buy an adaptor from your local hi-fi shop because miniature jack plugs are all that the ZX81 comes equipped with.

To connect the ZX81 to your tape recorder, simply plug the two jack plugs on the twin audio lead that came with your machine (you can easily recognise it because it's the only lead not already in use by this point!) into the two sockets marked 'MIC' and 'EAR' on the side of your ZX81. It doesn't at this stage matter which plug goes into which socket. Next plug the same colour jack plug that is in the MIC socket into the microphone socket on your tape recorder. You

are now ready to record your first program.

To give the ZX81 something to save, type in:

```
10 REM THIS IS A TEST
20 REM THIS IS THE SECOND LINE
30 REM THIS IS THE THIRD LINE
```

and then type

```
SAVE "TEST"
```

At this point your ZX81 is all set to record a program on tape so the next thing to do is to set the tape recorder running. Insert a blank tape, set the record level to about three-quarters of its maximum volume (this is unnecessary if your recorder has an automatic volume control) and set the tone control (if your recorder has one) to a fairly high level – that is away from bass, towards treble – and set it recording. When you are ready to record, press NEWLINE and you will see a pattern of horizontal lines appear on the screen – this means that the ZX81 is recording a program. When the message ‘ \emptyset/\emptyset ’ next appears on the screen the program is saved on tape. However, it is still possible that although the program is recorded on tape it may not be good enough to be read back into the ZX81 for one of a number of reasons. For example, you may have the record level set too high or there may be a fault on the tape.

To check that the program has been successfully saved, rewind the tape, unplug the jack plug from the microphone socket on the tape recorder and plug in the *other* jack plug into the earphone socket. Now type

```
LOAD "TEST"
```

or simply

```
LOAD ""
```

The first version of LOAD will search the tape until a program with the name “TEST” is found but the second version will load the first program on the tape irrespective of its name. Both these commands will have exactly the same effect in this case. Having typed one or the other of these commands, press NEWLINE and set the tape recorder running and if everything has gone to plan the next message that you should see on the screen will be another ‘ \emptyset/\emptyset ’ meaning that the program recorded on tape can indeed be read. If this message does not appear then something has gone wrong. For some reason the ZX81 has misread the information on the tape or cannot even find it.

In this case the best thing to do is to wind the tape back and listen to it (taking the jack plug out of the earphone socket on the recorder in order to hear what is happening). At the start of the tape you will hear a steady tone. This should be loud but not distorted. If this sounds OK then wind the tape on and listen to an unrecorded patch at the same volume. You should hear a very soft hiss. If it sounds like a rain storm then reduce the volume and perhaps alter the tone control and try again. If an unrecorded patch gives absolute silence then you should try again with the play-back volume increased.

Don't worry if you have to spend a bit of time setting up your tape recorder to accept programs from your ZX81. If you continue to have problems try experimenting with different brands of tape. In general, a low-noise tape is recommended but it depends to some extent on the quality of your tape recorder. Loading pre-recorded tapes (even commercially available ones) can sometimes be a problem. One hint that can speed up the process of finding the correct levels at which to set the controls on your recorder is to wind to the middle of the program, type `LOAD` followed by `NEWLINE` and then slowly increase the volume of the recorder, and also alter the tone control, until the TV screen displays four or five thick black bands. Once you've found the correct volume and tone levels to produce this effect you should have no difficulty when you rewind and try to load the program from its beginning.

Once you have the tape system working, don't alter the volume controls or tone controls unless it is necessary. And remember that it is a good idea to change the jack plugs around according to whether you are saving programs or loading from tape. Plug in the microphone socket for `SAVE` and a plug in the earphone socket for `LOAD`.

A test card program

At this stage you should feel confident enough to tackle entering a longer program. The program given below will produce the simple test card pattern shown in Fig. 2.4. Apart from providing a good test of your ability to use the keyboard, as it uses all the possible combinations discussed above, you could use it to adjust the TV set that you are using with your ZX81 to produce the sharpest possible image with the best contrast. At this stage, you are not expected to understand how this program works – it uses commands that will be



Fig. 2.4. A simple test card pattern.

explained later in the book. Moreover, it is not meant to be any more than a simple display – just sufficient to give you some lines, words and graphics characters with which to experiment with the controls of your TV. To keep things simple, it is a 1K ZX81 program and so if it were any longer or more complicated it would run out of memory. Because the screen display uses so much memory, the pattern does not completely fill the screen – so don't try adjusting your TV set to stretch it! Before you start typing in the program the following notes might help, but first type NEW to clear the ZX81's memory.

Every word, apart from 'ZX81 TV' in line 140 and 'CHANNEL 36' in line 150, is entered using a single key, so search the keyboard until you find the word! Because of the difficulty in printing the characters which are entered in graphics mode, a special notation is used. Wherever you see a character in square brackets this indicates that you should enter the character in graphics mode. If the character is preceded by an up-arrow (^) the press SHIFT at the same time. So, for example, the [^ E] in line 10 means, enter graphics mode and press E while holding SHIFT. The result should be a black corner piece. Remember to leave graphics mode (by pressing SHIFT and 9 again) whenever the second of a pair of square brackets is encountered. Notice that the word ZX81 is to be displayed in reverse graphics – white on a black background. To achieve this effect you enter graphics mode prior to typing the Z and leave it after typing the 1. The black block character in line 150 is produced by typing a space in graphics mode. Line 5 has no effect whatsoever on the program. It is just a note to the programmer to remind you what the program is intended to do. You can leave it out if you prefer but when you write

programs of your own that you want to keep for later use it is good practice to give them *titles* in this way.

```

5 REM TESTCARD
10 PRINT "[ ^ E]";
20 FOR I=1 TO 30
30 PRINT "[ ^ 7]";
40 NEXT I
50 PRINT "[ ^ R]"
60 FOR I=1 TO 14
70 PRINT "[ ^ 5]";TAB 31;"[ ^ 8]"
80 NEXT I
90 PRINT "[ ^ W]";
100 FOR I=1 TO 30
110 PRINT "[ ^ 6]";
120 NEXT I
130 PRINT "[ ^ Q]";
140 PRINT AT 5, 10; "[ ^ A][ZX81] TV [ ^ A]"
150 PRINT AT 7, 11; "[SPACE] CHANNEL 36[SPACE]"

```

Fig. 2.5. shows what the above program looks like typed up on the screen, before you press RUN.

```

5 REM TESTCARD
10 PRINT "[ ^ E]";
20 FOR I=1 TO 30
30 PRINT "[ ^ 7]";
40 NEXT I
50 PRINT "[ ^ R]"
60 FOR I=1 TO 14
70 PRINT "[ ^ 5]";TAB 31;"[ ^ 8]"
80 NEXT I
90 PRINT "[ ^ W]";
100 FOR I=1 TO 30
110 PRINT "[ ^ 6]";
120 NEXT I
130 PRINT "[ ^ Q]";
140 PRINT AT 5, 10; "[ ^ A][ZX81] TV [ ^ A]"
150 PRINT AT 7, 11; "[SPACE] CHANNEL 36[SPACE]"

```

Fig. 2.5.

When you have finished entering the program, simply press RUN and you should see the test card pattern appear on the screen. If it doesn't look like Fig 2.4 then you are likely to have made a typing error so check the whole program very carefully against the listing. Press the LIST key so that the program is displayed on the screen. You may have omitted a line. This is easy to remedy. To insert a line

just type it, with its line number, and it will appear at the bottom of the screen. Press NEWLINE and it will automatically assume its correct position in the program listing. Numbering in computer programs is normally done in jumps of ten so that you can easily insert extra lines if you need to. If you want to delete an entire line from a program just type its number and press NEWLINE and the line will automatically disappear – so do be careful not to delete program lines you want in this way.

If you find an error in a line you do not have to retype the entire line, thanks to the ZX81's editing facility that we have already discussed. Now is the time to check that you can use it to good effect. With the program listed on the screen, if you use the up-arrow and down-arrows (7 plus SHIFT and 6 plus SHIFT respectively) you will find that you can move the cursor up and down the screen to any line that you desire. Suppose that in line 60 you have typed 41 rather than 14. When you list the program, the cursor is displayed at the beginning of the program – in this case at line 5. You will have to press the down-arrow six times for it to reach line 60, pausing for the display to adjust itself each time. Once you reach the line with the error in it and then press EDIT (1 plus SHIFT), you will find that it appears back at the bottom of the screen (the *input area*) where you first entered it. You can now use the right- and left-arrow keys (on 8 and 5) and the RUBOUT key (0 plus SHIFT) to edit it and hopefully correct any mistakes. So to change 41 to 14 you would hold SHIFT down while you pressed the 8 key nine times. Keeping SHIFT down you would press RUBOUT. Using the SHIFT and 8 you could then move beyond the 1 that still remained on the line and then, having released SHIFT, you could type the 4 required to correct the line. When you have finished editing the line, simply press NEWLINE and the corrected line will take its rightful place back in the program.

Notice that you can sometimes save time when entering a program by making use of the editing facility. Lots of the lines in this program are very like each other. For example, line 20 and 60 are very similar. So, instead of entering each one from scratch, try using the edit facility. To enter line 60, move the cursor back to line 20 and then press EDIT (1 with SHIFT). This will copy the line back into the input area where you can change the line number to read 60 – to do this you have to delete the 2 of 20 and replace it by 6 – and then move the cursor to the end of the line. Delete the number 30 (SHIFT and the 0

key) and then type in 14. If you then press NEWLINE you will find that line 60 appears in the program without your having had to type it all in. You may also like to use the editing facility for copying from, say, line 10 to line 30. If so, remember that you have to enter graphics mode and press SHIFT down while entering the new graphics symbols.

Connecting the ZX Printer

You may think that it is only worth investing in a printer once you've written some useful programs – ones that will produce results for which you would want to have a permanent record. However, the ZX Printer really comes into its own as a programmer's aid. For one thing, a printer can be useful for keeping a record of programs that you have written but don't want to store on cassette tape. Moreover, if you are working on a long program it can be a great help to be able to inspect the whole program at once – rather than in screen-size chunks. Connecting the ZX Printer to the ZX81 is extremely simple. The first step to take is (if necessary) to replace the small power supply that came with the original ZX81 with the 1.2 amp supply that arrived with the printer. Plug the power supply into the side of the ZX81 as usual. Once you've plugged in the new, bigger power supply there is no need to revert to the smaller one. Indeed, installing it should make your ZX81 more reliable if anything, so you can put the old one away or in store. Turning to the printer itself, the blue connector should be pushed firmly into the slot at the rear of the ZX81. Notice that there is a slit in the printed circuit board that is exposed in this slot and a solid key in the third position of the blue connector. These fit together to prevent you from making the connection the wrong way round.

It is important to maintain the connection between the ZX81 and the printer. If you accidentally jog the printer or the ZX81 and disturb their intercommunication you may lose your program – even if the printer is not actually in active use. The best advice is to place both the ZX81 and the printer on a firm surface such as a table.

There are three ZX81 commands that control the printer. LLIST is the equivalent of LIST – it causes the current program to be listed to the printer rather than to the TV screen. Similarly, LPRINT is the equivalent of PRINT while typing COPY causes the whole of the TV screen's current display to be 'dumped' to the printer.

You may be interested to know how the ZX Printer works. It uses an electrostatic process. Two conductive styli traverse the paper and wherever a black dot is required the aluminium coating is etched off by an electrical pulse to reveal the black backing. Having two styli increases the speed of operation. The versatility of the printer – particularly its ability to reproduce screen displays – is due to there being extra software incorporated into the ZX81 itself.

Extra RAM

As mentioned in Chapter 1, you can go a surprisingly long way with the ZX81's 1K of RAM. However, if you do want to write programs of any length or complexity, or want to handle a lot of data, then you'll need to add extra memory. We use the Sinclair ZX 16K RAM pack but there are other 16K packs on the market and also 32K and 64K packs and their installation is the same in practice. The RAM pack is connected via the same slot as the printer and can either go in place of the printer or, if you want to use your printer as well, can be plugged into the slot at the other side of the printer connector. If you inspect this slot you will see that it is identical to that on the rear of the ZX81 itself. The advice given above about taking care that the connection to the ZX81 is constantly maintained applies equally to the RAM pack. The ZX81 is sensitive to the slightest disturbance. It is possible to make a permanent connection, soldering the RAM pack to the ZX81. Alternatively, you can use strips of velcro or a strip of an adhesive material such as *Blutack* to secure the connection. Or, you can mount the two pieces of equipment on a board so that they cannot move in relation to one another. We have experienced no problems when using them together on a firm, level surface. However, it is best to find a suitable method of guaranteeing a reliable connection before you embark on writing long programs that you would be upset about losing.

As you go along

This chapter has dealt with setting up your ZX81 system preparatory to use, and with familiarising yourself with the keyboard and the rudiments of entering programs. As you learn BASIC and the special features of the ZX81 from the rest of this book, you are bound to improve both in your understanding and your use of the keyboard

until you can hardly remember what all the fuss was about! However, until then it is wise to recall the advice given at the start of this chapter and try not to let the frustration produced by typing errors interfere with your understanding of computing in general and BASIC in particular.

Chapter 3

First steps - Variables, PRINT, LET and INPUT

A program is a list of instructions that your computer can carry out. The question that this poses is what sort of instructions can you use in a computer program? It is clear that instructions like 'go and make a cup of tea' are too vague for anything other than a human to cope with! Instructions used in a computer program must be precise. They have to specify exactly what must be done and perhaps, less obviously, they have to specify what it has to be done to. In other words, a computer instruction tells the computer what to do and what to do it to. In this chapter we will look at the simplest objects in BASIC and some very simple things that you can do with them.

Variables

The idea of a *variable* is the most important single idea in programming. A variable is an area of computer memory that is used to store information. This sounds like an easy idea but it has one or two subtle points. If you are going to store information in an area of memory you are going to need some way of referring to it. You're going to have to give it a name! This is not such an unusual idea if you think about other, more traditional, ways of storing data. For example, each file in a filing cabinet is normally given a name that identifies it and it alone. Just think of the confusion of asking for a file if two files had the same name! It is just the same with BASIC; an area of memory that is used to store information, a variable, must be given a unique name that can be used to refer to it. The only additional difficulty with a BASIC variable is that you must also define what sort of objects you are going to store in the memory area. One reason for this – we will meet others later – is that the amount of memory set aside to store the information depends on its type. For the time

being the only sort of information that we will store in memory will be numbers of any type. A variable that is used to store a number is called a *numeric variable* or a *simple variable*.

You cannot give a variable any name that takes your fancy because this would lead to confusion on the computer's part. For example, suppose you gave the name '1' to a variable. How would the ZX81 know the difference between the variable 1 and the number 1? In the case of the ZX81 you can give a simple variable a name of any length as long as it starts with a letter and thereafter uses only letters and digits. You can also insert spaces anywhere in the name to make it more readable but the ZX81 ignores them. Here are some examples of simple variable names that are allowed:

```
SUM
TOTAL SCORE, TOTALSCORE
THIS IS THE LONGEST NAME THAT ANYONE WOULD
    EVER WANT TO USE
TOTAL1
DAY2MONTH3YEAR83
```

Notice that both the versions of 'TOTALSCORE' are treated as the same name. It is often difficult to think up names for variables that suggest the nature of the information to be stored in them but it is well worth doing. If you come back to read a program after a long time clear, obvious variable names can make it a lot easier to re-understand your own program! Even so, you should try to avoid very long variable names – they can be very boring to type out over and over again in a program. Some examples of names that the ZX81 would not allow are given below:

Name	Reason for rejection
1 DAY	starts with a number
*DATE	starts with an * which is not a letter
DATE*	contains * which is not a letter and is not a digit
ANSWER?	? is not a letter or a digit
OVER-TIME	- is not a letter or a digit

Storing things in variables – LET

Now that we know about variables and how to give them names it is time to discover how to store information in them. This can be done using the BASIC command LET. For example:

```
10 LET TOTAL=56
```

will store the number 56 in an area of memory called 'TOTAL'. If you recall, lines of a program are entered with line numbers that control their order in the list of instructions that make up the program.

This example is in fact our first program! If you enter it exactly as written nothing will happen until you enter RUN, when the ZX81 will start obeying the list of commands. In this case there is only one command and this is very easy to obey – the number 56 is stored in an area of memory called 'TOTAL'. If you think about it, just a little more than this has happened. Before the program was run there was no area of memory called 'TOTAL' to store 56 in! When the ZX81 comes across the name of a variable that you wish to use to store something in, it checks to see if it already exists and if it doesn't it sets aside an area of memory of the right size and remembers its new name. So this innocent single line program has two effects – it creates the variable called 'TOTAL' and then it stores the number 56 in it.

Finding out what's in a variable – PRINT

The one line example in the previous section is a little disappointing because we have to take on trust that the ZX81 has actually stored the number 56 in a variable called 'TOTAL'. What we need is a command that will make the ZX81 find the variable and print its contents on the TV screen. The BASIC command with this effect is, most reasonably, called PRINT. If you add a new line, numbered line 20, to the previous example you will have the following two-line program:

```
10 LET TOTAL=56
20 PRINT TOTAL
```

If you RUN this program you will be pleased to find that the number 56 is printed in the top left-hand corner of your TV screen.

It is important that at this point you understand exactly what is happening as a result of this two-line program. Later on when you have absorbed BASIC almost as a second language you will understand what is going on without even thinking about it, but for

now it is all too easy to read this two-line program and think you understand it because it *sounds* OK! So, to recap what we have already learnt, the first line creates a variable called 'TOTAL' and stores 56 in it. The second line finds the area of memory with the name 'TOTAL' and prints what is stored in it on the screen. (Notice that is easy for a beginner to think that 'PRINT TOTAL' would print the word 'TOTAL' on the screen. So, if you already understand why this interpretation is incorrect you are no longer a beginner!)

For the PRINT statement to work it has to be possible for the ZX81 to find the variable that it refers to. If for some reason you try to print a variable that hasn't been created then the ZX81 will, quite rightly, give you an error message. To see this, delete line 10 from the previous program (simply by typing 10 and NEWLINE) and RUN it. You should be able to understand why you get a 'variable not found' error message at the bottom of the screen. This is your first bug!

Arithmetic

Our programs are slowly becoming more interesting but they are still a long way from being useful. We can now store numbers in variables and print out what is stored in any variable but so what! To be of any use we have to be able to change what is stored in a variable and print out something that we regard as an answer. The key to doing this lies in the idea of an *arithmetic expression*. An arithmetic expression is nothing more than a piece of arithmetic that you haven't yet worked out. For example, $3+6$ is an arithmetic expression that works out or evaluates to 9. You can write an arithmetic expression on the right-hand side of the equals sign in a LET statement with the effect that the ZX81 will evaluate the expression and store the result in the variable. For example, try:

```
10 LET TOTAL=3+6
20 PRINT TOTAL
```

You will see 9 printed in the top left-hand corner of the screen. As promised, the ZX81 has evaluated the expression and stored the result in 'TOTAL'.

As with most things to do with computers, there are rules governing what makes a correct expression. You can use the four operations that you should be familiar with from simple arithmetic. Addition and subtraction are indicated by the usual symbols '+' and

‘-’ but multiplication and division use the symbols ‘*’ and ‘/’. The reason for using * to mean multiply instead of a cross is that the traditional symbol is too easy to confuse with the letter ‘X’. Some examples of correct arithmetic expressions are:

Expression	Evaluates to:
$3+2$	5
$3*2$	6
$6/2$	3
$3+2-4$	1
$2.1+3.3$	5.4

Apart from the four usual operations of arithmetic there are two others that can be used on the ZX81 – the unary minus and raise to a power. The unary minus sounds rather grand but it is simply the normal subtraction sign used in front of a single number. For example, the ‘-’ in $3-2$ is the normal subtraction sign but the ‘-’ used in -3 is the unary minus. Although the same sign is used in both cases, as we will see later they are treated slightly differently. The raise to a power sign is ‘**’. For example, $2**2$ is read as ‘two raised to the power of two’, i.e. two squared or four. The raise to a power sign is not used very often and it is mentioned here more for completeness than for its importance.

Understanding expressions – the order of evaluation

Although the idea of an arithmetic expression seems straightforward, there is a hidden complication. For example, if you write the innocent-looking expression $3+2*4$, does it mean ‘three plus two’ (i.e. five) ‘times four’ (answer:twenty) or does it mean ‘three plus’ the answer to ‘two times four’ (i.e. ‘three plus eight’, answer:eleven). It may seem strange to you that there are two possible ways to work out this expression because you may feel that one of the two methods is obviously correct and the other is equally obviously incorrect. However, even in arithmetic, there are no absolute answers! The correct interpretation is a matter of convention and isn’t something that is handed down from on high. The question of whether we do the

‘+’ or the ‘*’ first in an expression like $3+2*4$ is settled by a general agreement that multiplication is more important than addition and so it should be done first, making the correct answer eleven. This agreement that multiplication is more important than addition can be formalised in terms of assigning priorities to each operation and carrying out the operation with the highest priority first. The assignment of priorities can be extended to every operation that can be used in an expression (even some that we haven’t met as yet). The priorities that the ZX81 uses to sort out the order in which arithmetic should be carried out are:

Operation	Priority	
**	10	highest
Unary -	9	
*, /	8	
+, -	6	lowest

The reason why the priorities start at 10 and finish at 6 is to allow other operators that we have yet to meet to be assigned priorities. To evaluate an expression you should always work out the operators with the highest priority first. If two operators in an expression have the same priority then you should do the one furthest to the left first (i.e. in the absence of any other preference, you work from left to right).

All this may seem a little over-complicated just to carry out a little arithmetic but it is necessary if you want to write unambiguous expressions. However, there is another way of specifying the order of evaluation that can be used to override the usual priorities – brackets (). It is a long-standing convention that any parts of an expression enclosed in brackets are carried out first. For example, although $3+2*4$ is 11, $(3+2)*4$ is 20. If you’re ever in any doubt about how the ZX81 will evaluate an expression then put brackets around the parts that you want worked out first. Brackets sometimes waste time and effort but they can never cause trouble!

Variables and constants – the full expression

So far we have looked at arithmetic expressions involving only numbers

but there is no reason why we cannot use variables in expressions. If you write an expression such as 'TOTAL+3' the ZX81 will find the variable called 'TOTAL' and retrieve the number stored in it. It will then add three to this number. For example, if 'TOTAL' had 32 stored in it, the expression 'TOTAL+3' would evaluate to 35. Notice that there is no suggestion that what is stored in the variable 'TOTAL' is altered in any way. Its contents are simply used in the evaluation of an expression. A number such as 32 is known as a constant (because its value never changes!) and now we can see that an expression can be made up of variables and constants with the arithmetic operators +, -, /, *, **. An expression always evaluates to a constant and it is this constant that is stored in a variable by a LET statement.

A short program

Using all that we have found out so far about constants, variables and expressions we can now write a short program that adds two numbers together:

```
10 LET NUMBER1=23.34
20 LET NUMBER2=44.32
30 LET ANSWER=NUMBER1+NUMBER2
40 PRINT ANSWER
```

If you enter and RUN this program you will see that the sum of the two numbers in lines 10 and 20 is printed by line 40. Although this is another simple example, it demonstrates a wide variety of programming ideas. In lines 10 and 20 the by now familiar LET statement is used to store two constants in two variables. In line 30 the arithmetic expression 'NUMBER1+NUMBER2' is evaluated and the result is stored in a third variable 'answer'. Line 40 prints the contents of 'answer' on the screen. If you think this is easy, so far so good! Try changing lines 10 and 20 to add different numbers together and change line 30 to give you different arithmetic expressions.

Another way of altering variables - INPUT

In the previous example, the two variables 'NUMBER1' and 'NUMBER2' had numbers stored in them by use of the LET statement. This is convenient unless we want to use the program many times with different values. As suggested, the only way that it is

possible to change the values stored in the variables is to edit each line before running the program. Obviously what we need is a statement that will allow us to enter any value into the variable while the program is running. This is what the BASIC statement INPUT is for. For example, try the following program:

```
1Ø LET NUMBER1=5
2Ø INPUT NUMBER2
3Ø LET ANSWER=NUMBER1+NUMBER2
4Ø PRINT ANSWER
```

When you run this you might be surprised to find that nothing happens! Don't panic! What has happened is that line 1Ø was carried out and 5 was stored in the variable 'NUMBER1'. Then the ZX81 moved on to line 2Ø where it obeyed the command INPUT by waiting for you to type a number, and this is why nothing is happening. The ZX81 is waiting for you to type a number and then press NEWLINE to signal that you have finished typing/correcting the number. It then stores the number that you have typed in the variable 'NUMBER2' and proceeds to the next instruction. So if you haven't already done so, run the program and type in a number of your choice. You will be pleased to see your number with five added to it printed in the usual place.

We now have two ways of storing numbers in a variable – LET and INPUT. It is important to understand the difference between the way LET and INPUT work. As in the case of LET, if a variable doesn't exist before its use in an INPUT statement the ZX81 will create it. If you always want to store the same value or the result of an expression in a variable then use a LET statement. If you want to store a different value in a variable each time the program is run then use an INPUT statement.

Variables and constants as expressions

One of the most powerful features of BASIC is the way that almost anywhere that you can use a constant or a variable you can use an expression as well. For example, in the PRINT statement you can write:

```
3Ø PRINT NUMBER1+NUMBER2
```

and the ZX81 will evaluate the expression and print the result.

It is also true that the simplest forms of an expression are the constant and the variable. For example, the number 3 can be thought of either as a constant or as an extremely simple expression. Similarly, the variable 'TOTAL' can also be thought of as an expression. So not only can you use an expression wherever you might use a variable or a constant, you can also use a variable or a constant anywhere that you can use an expression! For example:

```
LET NUMBER1=NUMBER2
```

and

```
PRINT 3
```

are both valid BASIC statements.

Describing BASIC

It is difficult to describe any language and BASIC is no different. The trouble is that while it's easy to give an example of what is correct, it is difficult to explain all the possible correct variations. For example, at the start of this chapter the LET statement was introduced by

```
LET TOTAL=56
```

but this gave no hint that you could write things like

```
LET TOTAL=NUMBER
```

or

```
LET TOTAL=NUMBER1+NUMBER2
```

To try to overcome this difficulty it is usual to give a definition involving the general types of things that a statement allows. For example, the general form of the LET statement can be written as:

LET 'simple variable' = 'arithmetic expression'

where the things between the single quotes are not to be taken literally but replaced by an example of the stated type. So, in a real LET statement, 'simple variable' would be replaced by a variable name such as TOTAL, SUM, NUMBER etc.

Throughout the rest of this book BASIC statements will be introduced by examples and then defined in the same way as the LET statement above. As we learn more about a statement it may prove

necessary to redefine it to include a wider range of features. So far the two other BASIC statements that we have introduced, PRINT and INPUT, can be defined as follows:

PRINT 'arithmetic expression'

and

INPUT 'simple variable'

but as we shall see later these are not the final definitions!

PRINT prompting

Although we can now use INPUT to store information in variables, the way the ZX81 just stops and waits for you to type a number is a little unsatisfactory. What is required is the ability to print a message on the screen saying something like 'TYPE IN A NUMBER NUMBER NOW' or 'WHAT IS YOUR NUMBER'. Such a message is often called an input prompt. Try the following short program:

```
10 PRINT "WHAT IS YOUR NUMBER?"
20 INPUT NUMBER
30 PRINT "YOUR NUMBER IS ";NUMBER
```

Line 10 will display 'WHAT IS YOUR NUMBER' in the top left-hand corner of the screen and line 30 echoes your number after you have input it. In both cases the characters printed on the screen are the ones inside the double quotation marks. A set of characters in double quotes is known as a *literal string* or simply as a *string*.

Mixed PRINT

The ability to print messages on the screen is clearly a very useful facility for things other than just printing prompts. For example, consider the following program:

```
10 PRINT "WHAT IS YOUR FIRST NUMBER ?"
20 INPUT NUMBER1
30 PRINT "WHAT IS YOUR SECOND NUMBER ?"
40 INPUT NUMBER2
50 PRINT NUMBER1+NUMBER2
```

It would obviously be better to print a message saying that the

number about to be displayed was the sum of the two numbers. You can, in fact, use a single PRINT statement to print more than one thing at a time. For example, change line 50 in this program to

```
50 PRINT NUMBER1;" + ";NUMBER2;" = ";
   NUMBER1+NUMBER2
```

and you will see that the contents of 'NUMBER1' are printed, then a space and a plus sign followed by another space then the contents of 'NUMBER2' followed by an equals sign with a space on either side of it and the answer. You can consider this PRINT statement as a list of items to be printed, each item in the list being separated by a semicolon and printed in the next free printing position. Our general definition of PRINT can now be updated to read:

PRINT 'PRINT LIST'

where 'PRINT LIST' is a list of items separated by semicolons. The items can be either expressions or strings. Each PRINT statement starts printing on a new line. In Chapter 5 we will return to the definition of the 'PRINT LIST' and expand it to include ways of formatting the information on the screen, but this simple version of the 'PRINT LIST' will satisfy all our requirements until then.

Some sample programs

Even with so little BASIC it is possible to write some useful, if simple, programs. For example, if you want regularly to work out how many dollars you would buy for a number of pounds, then a currency conversion program would be useful. The overall outline of this program is easy to explain as follows:

- Ask the user for the conversion rate.
- Ask for the number of pounds to be used to buy dollars.
- Print the number of pounds times the conversion rate.

By now you should realise that this program is a simple one for your ZX81, so before you look at the version given below, try to write your own version. Remember to include input prompting and explanatory messages. There is no one perfect way to write any program, so don't worry if your version is different to the one given. It could well turn out to be better.

```

10 PRINT "POUNDS TO DOLLAR CONVERSION"
20 PRINT "WHAT IS THE CONVERSION RATE?"
30 INPUT RATE
40 PRINT "HOW MANY POUNDS DO YOU WANT TO
  SPEND?"
50 INPUT AMOUNT
60 PRINT
70 PRINT "FOR ";AMOUNT;" POUNDS "
80 PRINT "YOU CAN BUY ";RATE*AMOUNT;
  " DOLLARS"

```

Line 10 simply prints a title for the program. Lines 20 and 40 prompt and lines 30 and 50 accept the necessary input. Notice how line 60 is used to print a blank line to space the output into an input and a result section. Lines 70 and 80 print the answer with some explanation.

As another example, consider the problem of calculating the stopping distance of a car travelling at any speed in miles per hour. The main problem in writing this program is knowing how to calculate the stopping distance. It is important to realise at this stage that no computer can calculate something unless you can explain to it how to do the calculation. Looking at the Highway Code reveals that the stopping distance is made up from two components – a thinking distance and a braking distance. The thinking distance in feet is roughly the same as the speed in m.p.h. The braking distance is a little more complicated and is given by the square of the speed divided by 20. However both of these quantities are very easy for the ZX81 to calculate so to add to the usefulness of this short program it is reasonable to print out the thinking distance, the braking distance, the overall distance and how many car lengths it takes to come to a stop. This last calculation is based on the fact that the average (UK) car is 14 feet long. The program is now easy to write:

```

10 PRINT "STOPPING DISTANCE"
20 PRINT
30 PRINT "SPEED IN MPH = "
40 INPUT SPEED
50 PRINT "AT ";SPEED;" MPH"
60 PRINT
70 PRINT "THINKING DISTANCE= ";SPEED;" FEET"
80 LET BRAKDIST=SPEED*SPEED/20

```

```

90 PRINT "BRAKING DISTANCE= ";BRAKDIST;" FEET"
100 PRINT "OVERALL DISTANCE= ";SPEED+
    BRAKDIST;"FEET"
110 PRINT
120 PRINT "WHICH IS ";(SPEED+BRAKDIST)/14;" CAR
    LENGTHS"

```

Line 10 simply prints a title for the program. Line 30 prompts for the only information needed by this program – the speed in m.p.h. This is stored in the variable 'SPEED'. Lines 50 and 60 start giving the answer to the user by printing the speed that the calculation is for and leaving some space. Line 70 prints the thinking distance, which requires no calculation as it is numerically the same as the speed in m.p.h. The braking distance is calculated by line 80 and printed by line 90. The overall distance is calculated and printed by line 100. Notice that this is an example of using an expression in a PRINT statement. Line 120 calculates and prints the overall stopping distance in terms of car lengths. Notice that you have to put brackets around the addition to make sure that it is carried out first.

The final example in this chapter is a sizeable and really useful program if you are interested in DIY. A very common problem is that of estimating how much sand, aggregate and cement you should buy to cast a slab of concrete. To help with this difficult task the following program is a 'Concrete Calculator'. Once again, our first problem is knowing how to do the calculation before we begin writing the program. Looking up the relevant information in a book on building reveals the following: 1 cubic metre of concrete made up from 1 part cement, x parts of sand and y parts of aggregate (i.e. a 1:x:y mix) needs

$$\frac{1}{0.025*(1+x+y)} \text{ bags of cement}$$

and

$$\frac{x*1.5}{(1+x+y)} \text{ cubic metres of sand}$$

and

$$\frac{y*1.5}{(1+x+y)} \text{ cubic metres of aggregate.}$$

It's not too important to understand why these equations work, it is often the case that a program is written from information that the

programmer doesn't fully understand – and why should it be otherwise! Converting this information into a program is once again relatively easy. The first part of the program should ask for the dimensions of the concrete slab and then calculate its volume. The program should then ask for the mixture ratio and using the equations given above, work out the number of bags of cement and the volume of sand and aggregate required. Finally, the results should be printed out in a form that the user will find acceptable. The details of the program are:

```

10 PRINT "CONCRETE CALCULATOR"
20 PRINT
30 PRINT "WHAT IS THE THICKNESS IN MM?"
40 INPUT THICK
50 PRINT "WHAT IS THE LENGTH IN M?"
60 INPUT LEN
70 PRINT "WHAT IS THE WIDTH IN M?"
80 INPUT WIDTH
90 PRINT
100 LET VOL=THICK*.001*LEN*WIDTH
110 PRINT "TOTAL VOLUME = ";VOL;" CUBIC M"
120 PRINT
130 PRINT "HOW MANY PARTS OF SAND TO ONE OF
    CEMENT?"
140 INPUT PARTSAND
150 PRINT "HOW MANY PARTS OF AGGREGATE TO
    ONE OF CEMENT?"
160 INPUT PARTAGG
170 LET TOTAL=1+PARTSAND+PARTAGG
180 LET CEMENT=VOL/TOTAL/.025
190 LET SAND=VOL*PARTSAND*1.5/TOTAL
200 LET AGG=VOL*PARTAGG*1.5/TOTAL
210 PRINT "USING A 1:";PARTSAND;" ";PARTAGG;
    " MIX"
220 PRINT "YOU NEED –"
230 PRINT CEMENT;" BAGS OF CEMENT"
240 PRINT SAND;" CUBIC M OF SAND"
250 PRINT "AND ";AGG;" CUBIC M OF AGGREGATE"

```

Lines 30 to 110 ask for the dimensions of the slab and both calculate and print the total volume. Lines 130 to 160 ask for the ratio of the

mix and line 170 calculates $1+x+y$ used in all of the following calculations. The most interesting lines of the whole program are 180 to 250. The first few lines (180 to 200) carry out the main calculations and the last section (210 to 250) prints the results. If you look carefully at the calculations you will see that an arithmetic expression in BASIC doesn't always look like the equation that it comes from. For example, you might fall into the trap of writing

$$\frac{1}{0.025*(1+x+y)}$$

as

$$1/0.025*(1+x+y)$$

but the result of this BASIC arithmetic expression is given by dividing 1 by 0.025 and then multiplying the answer by $(1+x+y)$. i.e. it works out as

$$\frac{1}{0.025}*(1+x+y)$$

rather than the equation that we are interested in. The correct expression is:

$$1/0.025/(1+x+y)$$

or if you want to use extra brackets to clarify matters:

$$1/(0.025*(1+x+y))$$

In the section that prints the results notice the way that each answer is *embedded* in the printed message. Remember that you do not have to write programs that produce results in a standard way – experiment with what looks good and seems natural.

Chapter 4

Looping And Choice - The Flow Of Control

So far we have written programs that are a list of instructions that are carried out one at a time from the top to the bottom. Although it is possible to write useful programs using nothing more, programming really only becomes interesting when you can change the order in which instructions are carried out.

The flow of control

If you look at any of the example programs at the end of Chapter 3 it should be possible for you to follow through with your finger the order that the instructions would be obeyed by the ZX81. You can think of this as tracing the *flow control* through the program. Each instruction has its turn at governing or *controlling* what the ZX81 is doing and it then *passes control* to the next instruction. In the absence of any other information the next instruction is taken to be the next line down. So the *default* flow of control is a line starting at the top of the program and finishing at the bottom. The following is a very simple program that demonstrates this default condition.

```
10 INPUT A
20 PRINT A
30 INPUT B
40 PRINT B
```



Fig. 4.1.

Looping – the GOTO

BASIC provides a single statement for changing the default flow of control – the GOTO statement. Try the following program:

```
10 LET TEST=0
20 PRINT TEST
30 GOTO 20
```

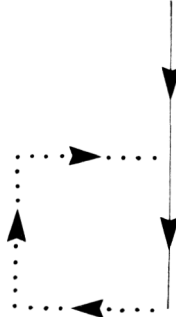


Fig. 4.2.

If you run this program you will see the screen fill with zeros (one to each line). The ZX81 then stops with the message '5/20'. This is in fact an error message informing us that, although the program has not finished, the screen is full. If you type the direct command CONT (telling the ZX81 to continue) you will see the screen fill with zeros once more and the same error message reappear.

This program is our first example of a *loop*. Tracing the flow of control through the program soon shows why the word loop is appropriate. First line 10 stores zero in the variable 'TEST', then line 20 prints the contents of 'TEST'. Line 30 is new in that it uses GOTO but its meaning should be clear from just reading it. The statement GOTO 20 causes the ZX81 to obey line 20 as the next instruction. So after line 30 control is transferred to line 20 and the contents of variable 'TEST' are printed on the screen for a second time. After this, line 30 is again carried out. This transfers control back to line 20 and so on until the screen is full of zeros. At this point the ZX81 wants to display another zero but can't do so as it has run out of space. The program calls for the repetition of lines 20 and 30 forever and tracing the flow of control shows it to take the form of a loop. It is not a very useful loop, however, and because it has no predetermined stopping point (it is only the ZX81's limited screen size that prevents it from going on forever) it is usually called an *infinite loop*.

A GOTO statement can be used to force the ZX81 to carry out any instruction next. Its general form is:

GOTO 'arithmetic expression'

so you can write things like

GOTO 2*10

(meaning the same thing as GOTO 20) but this sort of thing is not used very often and it is better to think of GOTO as:

GOTO 'line number'

You may be wondering what happens if you write something like GOTO 50 and line 50 doesn't exist. Most versions of BASIC would simply stop and give you an error message to the effect that you are trying to transfer control to a non-existent line but the ZX81's version of BASIC is a little different. If the line number doesn't exist then control is transferred to the line with the next highest line number. For example, if you use GOTO 50 and line 50 doesn't exist but line 55 does, then control will pass to line 55. If there is no 'next highest line', i.e. the GOTO tries to transfer control out of the program, then the program stops without an error message. This means that it is virtually impossible to get an error by using a GOTO! You may think that this is an advantage but take great care because if you have made a typing error in a GOTO the ZX81 won't tell you that you are transferring control to a line that doesn't exist. This might not cause any problem at first because the next highest line number might just be the line you wanted to GOTO anyway, but if you insert any lines of program later on trouble might appear from nowhere!

Although the example of the infinite loop serves to introduce the idea of transferring control to a different point in the program, it doesn't really indicate the sort of thing that a loop is used for. An important idea in programming is the repeating of series of operations. This is often called *iteration*. For example, the instruction LET COUNT=COUNT+1 simply adds one to the contents of the variable 'COUNT' and stores the answer back in 'COUNT'. In other words it increases the number stored in 'COUNT' by one. If you repeat this operation by using a GOTO you have something more than adding one to a variable – you have a program that counts! Try the following:

```
10 LET COUNT=0
20 LET COUNT=COUNT+1
30 PRINT COUNT
40 GO TO 20
```

You will see the screen fill with the numbers 1 to 21 and then the familiar error message will appear. If you type CONT time and again, you can keep the numbers coming for a very long time! Notice that by adding a GOTO to an instruction that adds one to a variable we actually seem to produce a program that does a bit more than just count. In fact, we generate a *sequence* of numbers. This is much more the flavour of real programming than the one-after-the-other programs in Chapter 3. To see looping doing something a little more useful try:

```
10 LET COUNT=0
20 PRINT "X=" ;COUNT;"XSQUARED=";COUNT*COUNT
30 LET COUNT=COUNT+1
40 GOTO 20
```

This program will print out two lists of numbers, the second being the square of the first.

Even if you are very familiar with the idea of a loop you can be confused about what the current value of a variable is at any point in the loop. For example, in the case of the counting loop program, the first value of 'COUNT' that was printed was one but in the squares program the first value was zero. This difference is simply due to *where* in the program the line that adds one to 'COUNT' is placed and *also* what value 'COUNT' is set to before the loop starts. If you change line 10 in the squares program to read 10 LET COUNT=1 then the first value to be printed will be one.

Understanding what goes on in a loop gets easier with practice but it's all a matter of following through the action of the program clearly and without rushing.

Choice and conditions – the IF statement

Although the GOTO is a very useful statement, the only thing that you can do with it is to form infinite loops. This is fine for simple things like printing tables of values but is a bit too crude for many applications. What we are lacking is a statement that will stop the loop when a *condition* is satisfied. For example, suppose we want to write a program that will add ten numbers together and then print out the answer. At the moment the best that we can do is to read in each number in turn and add it to a *running total* which we print out each time, as in the following example:

```

10 LET TOTAL=0
20 LET COUNT=0
30 PRINT "NUMBER= ?"
40 INPUT NUMBER
50 PRINT "NUMBER = ";NUMBER
60 LET TOTAL=TOTAL+NUMBER
70 LET COUNT=COUNT+1
80 PRINT COUNT;"TOTAL = ";TOTAL
90 GOTO 30

```

If you run this program you will find that it produces rather a lot of output that you don't need. Indeed, it runs out of space before it finishes – type CONT when the message '5/20' appears to allow it to finish. What we really want to do is read in the ten numbers, keep a running sum and only print out the answer at the end. This can be achieved using the IF statement:

```

10 LET TOTAL=0
20 LET COUNT=0
30 PRINT "NUMBER = ?"
40 INPUT NUMBER
50 PRINT "NUMBER = ";NUMBER
60 LET TOTAL=TOTAL+NUMBER
70 LET COUNT=COUNT+1
80 IF COUNT=10 THEN GOTO 100
90 GOTO 30
100 PRINT "TOTAL = ";TOTAL

```

The only difference between this and the first program to add ten numbers together is the use of the IF statement in line 80. Each time through the loop formed by the lines from 30 to 70 the IF statement is obeyed. This takes the form of comparing the contents of the variable 'COUNT' to 10. If it isn't equal to 10, control passes to the next statement, i.e. line 90, and then GOTO following the THEN has no effect. However if 'COUNT' is equal to 10 the GOTO following the THEN is carried out and control passes to line 100 printing the answer and ending the loop.

There are many ways of using the IF ... THEN GOTO statement to alter the flow of control depending on whether or not a condition is true. Before we can go on to investigate the sort of thing that can be done with IF we have to find out what types of conditions we can use.

All of the conditions that you can use in an IF statement take a very simple form:

'arithmetic expression1' 'relation' 'arithmetic expression2'

We already know what an 'arithmetic expression' is, so the only new element is the 'relation'. In BASIC there are six relations:

Relation	Meaning
=	Equals.
>	Greater than.
<	Less than.
<=	Less than or equal to.
>=	Greater than or equal to.
<>	Not equal to.

Notice that on the ZX81 each of these symbols is entered by a single keypress. You will get an error if you enter a '<' and a '>' to make up a single '<>'. The meaning in BASIC of each of these relations is the same as their normal meaning.

Conditions are very often called 'conditional expressions'. A condition is like an expression in that it evaluates to a value but in a condition there are only two possible results, *true* or *false*. It is important to read conditions in the right way to avoid confusion. For example, the condition 'COUNT=3' is not an instruction to make COUNT equal to 3, it is a question about what is stored in 'COUNT'. If the value stored in 'COUNT' is 3, then 'COUNT=3' is true but if the value is anything other than 3 then 'COUNT=3' is false. Thus the ZX81 uses the sign '=' in two different ways – as an instruction to store a value in a variable and as a relation in a condition. The only way to tell which is the correct meaning in any case is to look at the rest of the instruction. Some examples of conditions are:

COUNT<>4	True if COUNT is 4, otherwise false.
COUNT*2>10	True if COUNT*2 is greater than 10, otherwise false.
3>1	Always true.
1>6	Always false.

To reinforce the idea that a condition is an expression that evaluates to one of two values (true or false) it is worth saying that the ZX81 represents both values as numbers. True is represented by 1 and false is represented as 0. In other words, if a condition is true it evaluates to 1 and if it is false it evaluates to 0. To prove that this is the case try the following program:

```
10 INPUT "FIRST NUMBER";A
20 INPUT "SECOND NUMBER";B
30 PRINT "THE RESULT OF ";A;">";B;" IS ";A>B
40 GOTO 10
```

You will find that either a 1 or a 0 will be printed depending on whether 'A>B' is true or false in the case of the numbers you typed. You might be surprised that you can write a condition in a PRINT statement where you would normally write an arithmetic expression. This is simply another reflection of the fact that a condition is an expression just like an arithmetic expression and it can be used anywhere that an arithmetic expression can. Indeed, you can mix conditional and arithmetic expressions with no problems as long as you pay attention to the order in which they are evaluated. (Relations have priority 4 which means that any arithmetic is done before they are evaluated). For example:

```
PRINT (1<2)+(3>1)+(3=2+1)
```

will print 2 on the screen because (1<2) is false and evaluates to 0, (3>1) is true and evaluates to 1 and (3=2+1) is also true and evaluates to 1, which gives 0+1+1, i.e. 2. (Notice that the conditions all have to be enclosed in brackets to stop the ZX81 trying to do the arithmetic first!) This sort of thing is fairly advanced BASIC so don't worry if you don't understand it fully – what is important is that you understand that something like 2=3 is an expression and evaluates to true or false rather than an instruction to do some operation.

We have taken a slight detour around the subject of the IF statement to examine the idea of a conditional expression, but armed with this new information the remainder of this chapter should seem easier. The general form of the IF ... THEN GOTO statement is:

```
IF 'conditional expression' THEN GOTO 'line number'
```

If the conditional expression evaluates to 1, or true, then the GOTO following the THEN is obeyed. If the conditional expression

evaluates to \emptyset , or false, then the statement following the IF is obeyed.

There are so many ways of using the IF ... THEN GOTO apart from breaking out of loops that it is difficult to give examples of everything. But if you understand the ideas of flow of control and the way that IF and GOTO can be used to change it then you should have no trouble in understanding the examples in the rest of this book.

Using IF

The only way to find out how useful IF can be is to write your own programs that use it. In this way you'll slowly pick up all the standard ways that you can change the flow of control depending on the result of a conditional expression. However, it might help to speed up this process and avoid clumsy ways of using the IF statement to give examples of some of the most common ways that it is used.

An IF statement can be used to skip a section of program:

```
1 $\emptyset$  INPUT A
2 $\emptyset$  IF A> $\emptyset$  THEN GOTO 4 $\emptyset$ 
3 $\emptyset$  LET A=-A
4 $\emptyset$  PRINT A
```

This short program refuses to let you enter a negative number! The IF statement in line 2 \emptyset checks to see if the number in 'A' is greater than zero and if so control passes to line 4 \emptyset – effectively skipping line 3 \emptyset . If 'A> \emptyset ' is false, line 3 \emptyset changes the sign of the contents of 'A'. In another program a different list of statements might be skipped according to some other condition. The 'shape' of the skip is depicted in Fig. 4.3 which illustrates how the result of the condition decides whether or not the list of instructions is carried out or skipped.

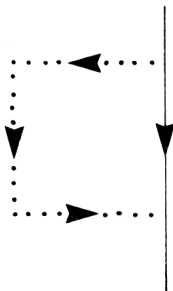


Fig. 4.3.

An extension of the idea of skipping some lines of BASIC is to choose between two different lists of commands. In this case it is easier to understand the idea after looking at the shape of the flow of control (see Fig. 4.4).

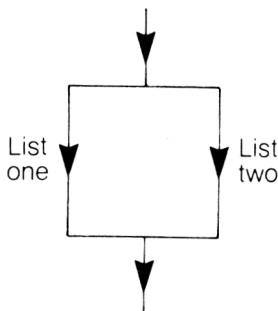


Fig. 4.4.

Which list is carried out depends on the condition used in the IF statement. The reason why it is useful to look at the flow of control diagram before looking at an example of an IF statement to select between two sets of instruction is that it is a little difficult to see the simple division into two in the BASIC. Consider the following program:

```

10 INPUT A
20 IF A<0 THEN GOTO 60
30 PRINT "A IS POSITIVE"
40 LET B=A
50 GOTO 80
60 PRINT "A IS NEGATIVE"
70 B=-A
80 PRINT A,B

```

According to the value of 'A', either lines 30, 40 and 50 are obeyed or lines 60 and 70. The division point in the diagram corresponds to the IF statement itself (line 20). The 'join up' point is line 80 because this is the first statement that will be carried out, no matter which of the two lists of instructions is selected. If you try to superimpose the flow of control diagram on the program you will see that, although it represents what happens, it is difficult to fit it. The reason for this is that it is impossible in BASIC to write the two alternative lists next to each other so the flow of control diagram is more like the one in Fig. 4.5.

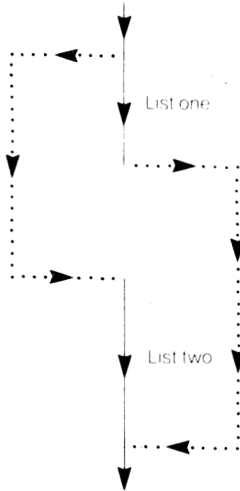


Fig. 4.5.

If you look at this diagram you should be able to see that it is a mangled form of the previous diagram. Using the IF to select between two alternatives is easier to understand from the first diagram but the second corresponds more closely to reality.

We have already met the only other important use of the IF statement, that of breaking out of a loop. The flow of control diagram for that circumstance can be visualised as shown in Fig. 4.6.

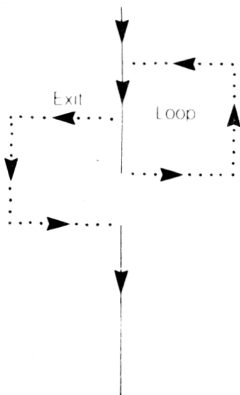


Fig. 4.6.

You should be able to see the familiar shape of an infinite loop. The path that leads out of the loop and back to the normal flow of control corresponds to a GOTO taken when the condition in the IF statement is true. Notice that the point at which the IF statement breaks out of the loop can be placed anywhere. In other words, you can break out of a loop anywhere from the first statement to the last. For example:

```
10 LET A=0
20 LET A=A+1
30 IF A=20 THEN GOTO 60
40 PRINT A
50 GOTO 20
60 PRINT "FINISHED"
```

has its exit point in the middle and

```
10 LET A=0
20 LET A=A+1
30 PRINT A
40 IF A=20 THEN GOTO 60
50 GOTO 20
60 PRINT "FINISHED"
```

has its exit point at the end of the loop.

In general, although it is possible to place the exit point of a loop anywhere, it is better to place it either right at the beginning or right at the end. The reason for this is that it is better to avoid carrying out part of a loop. (Technically, a loop with only one exit point placed at the beginning is known as a *while* loop and a loop with only one exit point at the end is known as an *until* loop. However, these names come from other computer languages and are not important when programming in BASIC.)

A loop that has its exit point at the end can be simplified by turning the condition round the other way, for example:

```
10 LET A=0
20 LET A=A+1
30 PRINT A
40 IF A=20 THEN GOTO 60
50 GOTO 20
60 PRINT "FINISHED"
```

can be turned into

```

10 LET A=0
20 LET A=A+1
30 PRINT A
40 IF A<>20 THEN GOTO 20
50 PRINT "FINISHED"

```

The loop in the first example comes to an end when 'A' is equal to 20 and line 50 contains the GOTO that otherwise makes the loop continue. In the second example, the GOTO has been eliminated by turning the condition around to make the loop continue when 'A' is *not* equal to 20. After seeing this sort of thing a few times you will adopt the shorter, neater form without thinking, but it is worth occasionally remembering where it derives from.

As stated at the beginning of this section, there are many ways of using an IF statement to alter the flow of control of a program and you shouldn't feel restricted to just those that have been introduced in this section. However, it is wise not to experiment too much because the more ways that you use the IF statement the more difficult your program will be to understand. In essence the rule is to make the flow of control diagram as simple as possible so that your programs will be easy to read, understand and debug.

The FOR statement

Apart from the position of the exit point, loops differ in two other ways. All loops continue until a condition is satisfied but in many cases this is equivalent to carrying out the loop a fixed number of times. For example, if you wanted to print the word 'HELLO' on the screen five times you could do it using:

```

10 LET A=1
20 PRINT "HELLO"
30 LET A=A+1
40 IF A<=5 THEN GOTO 20

```

However, this is such a common situation that BASIC provides two extra statements – FOR and NEXT – to make repeating lists of statements easier.

Using FOR and NEXT the program that prints 'HELLO' on the screen five times can be written as:

```

10 FOR A=1 TO 5
20 PRINT "HELLO"
30 NEXT A

```

The meaning of the FOR and NEXT should be clear from the program. The variable 'A' is used to count the number of times that the loop has been obeyed in the same way as in the earlier example. The difference is that everything is done automatically. The FOR statement first sets 'A' to one. Each time the NEXT statement is carried out one is added to 'A' and, as long as its value hasn't exceeded 5, control is transferred back to line 20 (i.e. there is an implied GOTO 20). The result is that the PRINT statement at line 20 is executed five times before control passes on to the statement following the NEXT. The general form of the FOR ... NEXT loop is:

```

FOR 'index variable'='start value' TO 'end value'
    .      .      .      .
    .      .      .      .
    .      .      .      .
NEXT 'index variable'

```

The 'index variable' is initially set to the 'start value'. Each time NEXT is reached, the 'index variable' is increased by one and, as long as the value hasn't exceeded the 'end value', control is transferred to the statement just after the FOR. The only restriction is that the name of the 'index variable' can only be a single letter – so 'index variable' can be any one of 'A' through 'Z'. To make sure that you understand exactly what a FOR loop can do, try the following examples:

```

10 FOR I=1 TO 10
20 PRINT I
30 NEXT I

10 FOR Z=100 TO 110
20 PRINT Z
30 NEXT Z

```

In general, both the 'start value' and the 'end value' can be full arithmetic expressions but it is important to realise that these are only evaluated *once* at the start of the loop. This becomes clear if you think of the FOR statement as only being carried out once at the start of the loop. The value of the 'index variable' can be used in arithmetic expressions during the loop *but its value must not be changed*. In other words, you can use the 'index variable' in a LET statement on

the right-hand side of an '=' sign but not on the left. For example, the following short program will print a multiplication table:

```

1Ø PRINT "WHICH TABLE – ENTER 2 FOR TWO TIMES
    TABLE ETC?"
2Ø INPUT T
3Ø PRINT "STARTING AT ?"
4Ø INPUT S
5Ø PRINT "ENDING AT ?"
6Ø INPUT E
7Ø FOR I=S TO E
8Ø PRINT I;" X ";T;" = ";I*T
9Ø NEXT I
    
```

Notice that both the start and end values of the FOR loop are arithmetic expressions, simple variables in fact! Also note the use of the 'index variable' 'I' in line 8Ø.

The simple FOR loop serves for most purposes, but there is a slightly more advanced form that is occasionally useful and is certainly worth knowing about. In the simple FOR loop, each time through the loop one was added to the value of the index variable. This is sensible if you are using the index variable to count the number of times that the loop has been carried out. However, it is sometimes the case that a calculation carried out inside the loop needs a value that changes by something other than one each time through the loop. This is catered for by the addition of the BASIC statement STEP, the general form of which is:

FOR 'index variable'='start value' TO 'end value' STEP
'increment'

The 'increment' specified following STEP is the amount that is added to the index variable each time though the loop. So the simple FOR loop is equivalent to STEP 1. The only thing that you have to be careful of when using FOR STEP is that you know when the loop will come to an end. The rule is that the loop terminates when the value of the index *exceeds* the 'finish' value. So the value of an index variable in a FOR loop can never become larger than the 'finish value'. To see this in action try the following examples:

```

1Ø FOR A=.4 TO 1Ø STEP .Ø1
2Ø PRINT A
3Ø NEXT A
    
```

or

```
10 FOR A=1 TO 100 STEP 25
20 PRINT A
30 NEXT A
```

The value of the *increment* following STEP can be negative, in which case it is better called a *decrement*. For example:

```
10 FOR A=100 TO 0 STEP -15
20 PRINT A
30 NEXT A
```

You may have some slight difficulty in working out when this and similar loops end. However, the rule is almost the same as for a positive increment. Each time through the loop the value of the index variable decreases by the increment and the loop ends when its value first drops below the 'end' value.

As well as having a negative increment, it is possible for either the 'start' or the 'end' value to be negative and this is where things can be confusing. Consider this short program:

```
10 FOR A=-100 TO 50 STEP 10
20 PRINT A
30 NEXT A
```

At first sight it may not seem to make sense. However, if you keep a cool head then you should be able to see that the same rules apply. The value of the increment is added to the index variable each time through the loop until its value exceeds the 'end' value if the increment is positive, or is less than the 'end' value, if the increment is negative.

Using the FOR loop

There are one or two rules that govern the use of FOR loops. As you can use any valid BASIC statement within a FOR loop it is possible to use a GOTO or an IF to leave a FOR loop before it is finished (i.e. before the index variable has reached the 'end' value). This is quite permissible in ZX BASIC but many other versions of BASIC will complain if you leave FOR loops in an unfinished state. Because of this it is better not to fall into the habit of using sloppy FOR loops.

It is often the case that FOR loops are used in combination. This is

easy to understand as long as you think logically about the way each loop works. For example:

```
10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A,B
40 NEXT B
50 NEXT A
```

This is correct because the FOR loop formed by lines 20, 30 and 40 is completely contained within the FOR loop starting at line 10 and ending at line 50. This means that the *inner* loop is carried out each time through the *outer* loop. It is all too easy to make a slight mistake and end up with:

```
10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A,B
40 NEXT A
50 NEXT B
```

which will give you an error message.

IF ... THEN

There is an even more general version of the IF statement than the one we have examined so far. As well as being able to follow the THEN by the BASIC command GOTO, you can in fact use any valid BASIC command. For example:

```
10 INPUT A
20 IF A<0 THEN PRINT "A IS NEGATIVE"
30 IF A=0 THEN PRINT "A IS ZERO"
40 IF A>0 THEN PRINT "A IS POSITIVE"
50 GOTO 10
```

The best way to think of this is as a sort of easier version of the IF to skip an instruction. In this case the instruction is only carried out if the condition is true. There isn't very much to add to this description except to emphasise the fact that you can follow THEN by any valid BASIC statement – including another IF!

This extended version of the IF statement seems very useful at first

but you quickly come to the conclusion that it's not often that you want to execute a *single* statement as a result of some condition.

There is a very simple BASIC statement that we have not discussed so far that is very useful when used in conjunction with the IF statement. Suppose that as a result of some condition the program should stop, then currently the only way that we know of achieving this is to GOTO a line number that is at the end of the program. The BASIC statement STOP can be used anywhere in a program to return control to the user. For example:

```
10 PRINT 'DO YOU WANT TO CONTINUE Y=1,N=0'
20 INPUT A
30 IF A=0 THEN STOP
40 IF A<>1 THEN GOTO 10
50 PRINT "I CONTINUE MASTER"
60 GOTO 10
```

Notice the use of STOP in line 30. Also notice how line 40 is used so that if a value other than 1 or 0 is input the original question will be repeated. If the user replies 1 then the program continues – in this case all it continues to do is to ask the same question until the user types 0, when it stops.

A final example

As a further example of both the IF and FOR statements, consider the problem of turning the stopping distance program given in the previous chapter into a sort of quiz. For a range of speeds the player is asked for the thinking, braking and total stopping distance in feet.

```
10 PRINT "STOPPING DISTANCE"
20 LET MARK=0
30 FOR S=10 TO 80 STEP 10
40 PRINT "WHAT IS THE THINKING DISTANCE AT";S;"
    MPH?"
50 INPUT T
60 IF T=S THEN LET MARK=MARK+1
70 PRINT "WHAT IS THE BRAKING DISTANCE AT ";S;"
    MPH?"
80 INPUT B
90 IF B=S*S/20 THEN LET MARK=MARK+1
```

```

100 PRINT "WHAT IS THE TOTAL STOPPING DISTANCE?"
110 INPUT D
120 IF D=S*S/20+S THEN LET MARK=MARK+1
130 NEXT S
140 PRINT
150 PRINT "YOU SCORED ";MARK
160 PRINT " OUT OF A POSSIBLE 24"

```

The flow of control summarised

It is difficult to say exactly what goes on in the mind of an experienced programmer. Whatever it is, it is a process that marks the difference between a beginner and an expert. One thing that does seem certain is that, in addition to the catalogue of programs that have been seen before and a collection of handy tricks, the standard forms of the flow of control diagrams that we have been studying in this chapter are ever-present. Is it worth mentioning at this point that it has been proved that you can write any program using only the default, select and conditional loop. This is so important that it is worth gathering together in, Fig. 4.7, the flow of control diagrams so that you too can store them away in your personal memory.

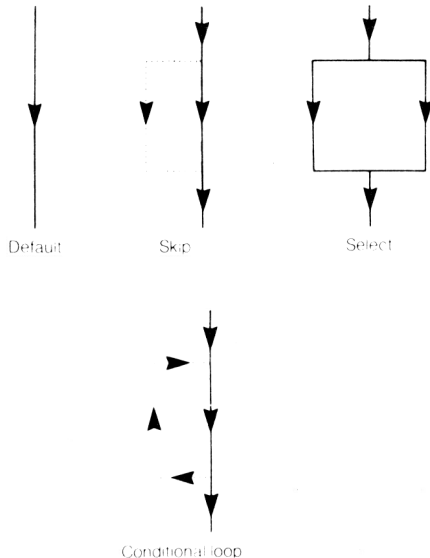


Fig. 4.7. Flow of control diagrams.

Chapter 5

Handling Text And Numbers

So far the only programs that we have written have used numbers. If this were all that computers could do they would be little different from pocket calculators! In this chapter we will look at how the ZX81 can handle characters and text just as easily as digits and numbers. In the last part of the chapter some other ways of extending the things we can do with data are introduced – arrays of numbers and strings.

Strings

In Chapter 3 the idea of a *string* – a collection of letters within double quotes – was introduced as a way of printing messages and prompts. In fact, what we have been calling a string is really a *string constant*. The use of the word constant might alert you to the fact that there are such things as *string variables*. A string variable is similar to a simple variable in that it is a named area of memory that can be used to store information. In this case, the information is a collection of characters instead of a number. The rules for naming string variables are different from simple variables. A string variable's name consists of a letter followed by a dollar sign. The dollar sign is used to distinguish between a simple variable and a string variable (e.g. B is a simple variable but B\$ is a string variable and therefore different). The LET statement can be used to store a string constant in a string variable and the PRINT statement can be used to print its contents.

```
1Ø LET A$="THIS IS A STRING"  
2Ø PRINT A$
```

In fact, a string variable can be used anywhere that a simple variable can as long as it makes sense. For example, you can use INPUT A\$ to store a string typed in from the keyboard while a program is running

but `LET TOTAL=3+A$` is obviously nonsense (you cannot add a string to a number!). Notice in particular the difference between

```
1Ø LET A$="1"
```

and

```
2Ø LET A$=1
```

Line 1Ø is fine because the 1 is enclosed in double quotes and is therefore a string but line 2Ø will give an error message because A\$ is a string variable and 1 is a number.

The introduction of string variables is exciting because it opens up the possibility of the ZX81 handling text and even dialogues. So far, however, the only sort of program that we can write is

```
1Ø PRINT "WHAT IS YOUR NAME ?"
2Ø INPUT N$
3Ø PRINT "HELLO";N$;" , I AM YOUR ZX81 COMPUTER"
```

which is all right for a start but it can result in unnatural dialogues like:

```
WHAT IS YOUR NAME ? FRED BLOGGS
HELLO FRED BLOGGS, I AM YOUR ZX81 COMPUTER
```

The trouble is that although we can INPUT, PRINT and store strings we have no way of changing them. This is rather like being able to INPUT, PRINT and store numbers but having no way of doing arithmetic – it's obvious that this would limit the programs that we could write! The answer lies in inventing an 'arithmetic' for strings so that as well as having arithmetic expressions we can use 'string expressions'.

String expressions

Before introducing the ZX81's facilities for handling strings it is worth considering what sort of things you might like to do and then see if the ZX81 can actually fit the bill. If a program had someone's first name in F\$ and their last name in L\$ then it would be useful to be able to *join* them together to form one longer string. Such joining together of strings is known as *concatenation*. Another thing that would be useful is the ability to *extract* part of a string. For example, you could extract the last name from a string consisting of an initial

and surname i.e. extract “BLOGGS” from “F.BLOGGS”. A string that is part of another string is often called a *substring*. For example, BLOGGS is a substring of F. BLOGGS. Another useful facility would be to replace a substring by another. For example, if we were trying to keep F. BLOGGS a secret we might want to replace the surname by astrisks giving “F. *****”. Finally, it would be a great advantage to be able to test for the presence of a particular substring in a string, for example, to see if the substring “BLOGGS” occurred in the name stored in N\$. To recap, the string operations that we would like to find are – concatenation, substring extraction, substring replacement and substring searching.

Our first requirement, string concatenation is immediately satisfied by the ZX81's concatenation sign ‘+’. If A\$ contains the string “ABCD” and B\$ contains “EFGH” then after

```
LET C$=A$+B$
```

C\$ will contain “ABCDEFGH”. Notice that we now have two uses for the symbol “+”, as the sign for addition and as the sign for concatenation. You can use the “+” sign more than once in a string expression. For instance,

```
LET C$=“MR ”+F$+“ ”+L$
```

will join up the four strings involved and if F\$ contains a first name “FRED” and L\$ a last name “BLOGGS” then C\$ will contain “MR FRED BLOGGS”. Notice the use of a single space between the two strings to avoid the result being “MR FREDBLOGGS”!

Extracting or changing a substring can be done by using a single new operation, *slicing*. Most other versions of BASIC use methods that are much more complicated than the ZX81's string slicing. Although ZX BASIC may be on its own when it comes to handling strings, in fact it offers a distinct improvement over other BASICs. A string slicer specifies a substring by giving the position of the first and last letters. For example,

```
PRINT “12345678” (3 TO 6)
```

displays the substring “3456” i.e. starting at the third character and ending at the sixth. Remember that ‘TO’ must be entered by a single keystroke (it is to be found on the 4 keypad). Typing the letter T and the letter O just won't do! The general form of a slicer is:

'String
expression' ('arithmetic
expression1' TO 'arithmetic
expression2')

and the substring that the slicer specifies starts at the character given by 'arithmetic expression 1' and ends at the character given by 'arithmetic expression 2' in the string that 'string expression' evaluates to. This may seem like a complicated definition, and indeed it does go further than the most common forms of the slicing notation found in other BASICs. For most of the time the 'string expression' is either a constant or a single string variable and the arithmetic expressions are again simply constant numbers or single variables, for example:

"ABCD"(2 TO 3)

or

"ABCD" (COUNT TO 3)

However, by now you may have realised that one of the most powerful principles in BASIC is that anywhere you can use a constant or a variable you are also allowed to use an expression. String slicing is no different in this respect and you can write things like:

("ABCD"+"EFGH")(START TO START+3)

which first concatenates the two strings "ABCD" and "EFGH" to form the single string "ABCDEFGH" and then extracts four letters, starting with the character at the position stored in 'START'. You shouldn't be frightened to write complicated string expressions any more than you would be over complicated arithmetic expressions.

There are a number of special cases of the slicing notation that are well worth knowing about. The start and end of a string are so often used in forming substrings by slicing that, if you leave 'arithmetic expression 1' out, the start of the string is assumed – and if you leave 'arithmetic expression 2' out, the end of the string is assumed. So:

"123456" (TO 5) means "123456" (1 TO 5) which is "12345"

"123456" (3 TO) means "123456" (3 TO 6) which is "3456"

"123456" (TO) means "123456" (1 TO 6) which is "123456"

There is also a very useful abbreviation that will extract a single character at any position in a string. Instead of having to write (n TO n) to extract the character at position n in the string you can simply write (n). So:

"123456"(3) means "123456" (3 TO 3) which is 3

It is possible to get things wrong when using slicing. For example, if you specify a character position that doesn't exist you will get an error code. For example, "123456" (4 to 7) will result in code 3, which indicates that the subscript is out of range; while "123456" (-1 TO 3) will produce error B since it involves a negative subscript. However, if you use a starting position that is less than the final position e.g. "123456" (3 TO 1) you might be surprised to discover that you do not get an error message. Instead, the answer is a very special form of string – the *null string*. The null string has no characters in it and plays a very similar role in string expressions to that of zero in arithmetic expressions. The string constant that corresponds to the null string is written "" i.e. a pair of double quotes with nothing in between. Notice the difference between "" and " ". The first is the null string and has no characters in it, the second is a string consisting of one character – a blank or space. From the point of view of a computer, a space is just as much a character as a letter of the alphabet; it takes up one position to print and it needs just as much computer memory to store! If you print a null string it has no effect whatsoever. The statements PRINT A;B and PRINT A;"";B produce the same result.

As an example of using slicing, consider the problem of printing the name of a month of the year given its number, that is, substituting DEC for 12 and MAY for 5 etc. Try the following short program:

```
10 LET Y$="JANFEBMARAPRMAYJUNJULAUGSEP
   OCTNOVDEC"
20 PRINT "MONTH NUMBER ?"
30 INPUT MON
40 PRINT "MONTH ";MON;" IS " ";Y$(1+3*(MON-1)
   TO 3*MON)
40 GOTO 20
```

If you enter a number in the range 1 to 12 the program will print the correct abbreviation for the month in question. The way that it works is by slicing out the three letter name from the long string Y\$. The best way of understanding the slicer in line 40 is by working it out by hand for a few values of 'MON'. If 'MON' is 4 then $(1+3*(MON-1))$ is 10 and $3*MON$ is 12 and (10 TO 12) specifies the three letters "APR".

Slicing can be used not only to extract a substring but also to change all the characters in the substring. You can use the standard

slicing notation to define a substring to be changed on the left-hand side of the “=” in a LET statement. For example,

```
1Ø LET A$="123456"
2Ø LET A$(3 TO 4)="ABCDEFGG"
3Ø PRINT A$
```

will print ‘12AB56’ on the screen. This feature is a very logical extension of the normal use of LET to store a string in a string variable. The slicer simply restricts the range of the characters that are altered by the LET. If the string on the right-hand side of the “=” in the LET is bigger than the substring specifies, the extra characters are ignored and, if it is shorter, then it is *padded* with blanks. For example,

```
1Ø LET A$="123456"
2Ø LET A$ (2 TO 5)="ABC"
3Ø PRINT A$
```

will print the string “1ABC 6” on the screen. (Notice the blank following the letter C.)

The only thing left from our initial list of string handling requirements is testing to see if a particular substring is present in another string. ZX BASIC doesn’t provide a direct method of achieving this but it does extend the use of conditional expressions (see Chapter 4) to strings and this can be used to achieve the same ends. You can use all of the relations that were introduced in that chapter with strings. The meaning of “=” and “<” are easy enough to understand. Namely, two strings are equal if they are of the same length and contain the same characters in the same order, otherwise they are not equal. However, what do the relations “<”, “>”, “<=” and “>=” mean when applied to strings? The answer to this question varies from BASIC to BASIC. In the case of the ZX81, however, they are defined so that A\$<B\$ is true if the string in A\$ would come before the string in B\$ in an alphabetically ordered list of strings. The trouble is that we are all so familiar with alphabetically ordered lists that we tend to forget how they work! If the two strings being compared are single letters, then A\$<B\$ if the letter in A\$ comes earlier in the alphabet than the letter in B\$. For example, “A”<“B” is true but “D”<“B” is false. What about comparing strings that contain single characters that are not necessarily letters, e.g. what do we make of “*”<“\$”? In the case of the letters, the alphabet provided

us with a ready-made order so what we need is to extend this order to include all the other symbols that the ZX81 can use. In other words we need a 'super-alphabet'! This is already available for the ZX81. If you look at Appendix A in the ZX81 Manual you will see a listing of the complete set of ZX81 characters in a predefined order and it is this that is used as the super-alphabet to decide if $A\$ < B\$$. If you don't have the ZX81 Manual to hand, or if you are just interested, you can print all the characters in their proper order by using the following program:

```
10 FOR I=0 TO 255
20 PRINT "CHARACTER ";I;" = ";CHR$(I)
30 NEXT I
```

Don't worry about the use of CHR\$ in line 20, its use of this will be explained in Chapter 6. If you run this program you might be surprised to see words like "PRINT" and "FOR" appearing on the screen as characters! This is simply a reflection of the fact that the ZX81 treats everything that can be entered as a single keystroke as a single character even if it appears on the screen as a word! You might also be puzzled by the appearance of so many question marks. Some of the characters in the ZX81 do not correspond to any printable shape – in other words, they are unprintable! For example, the "RUBOUT" character cannot be printed in the same way that the others can. In this case the ZX81 prints a question mark to indicate that it doesn't know what the character looks like!

Coming back to the question of whether or not $"*" < "$$ is true or false, $"*"$ is character 23 and $"$$ is character 13 so $"$$ comes before $"*"$ in the order and $"*" < "$$ is false. You can decide the truth or otherwise of any relation in the same way.

If the two strings contain more than one character, they are compared one character at a time until the first pair of different characters is found. The relationship between the two strings is then decided on the basis of those two characters. For example, $ABCD < AZCD$ is true because the first pair of letters that is different is B and Z and $B < Z$ is true. If one of the strings is the same as the other apart from the addition of a few extra characters then the comparison is based on length, thus $ABCD < ABCDEF$, because there is no pair of letters that is different and ABCD is shorter than ABCDEF.

The idea of the length of a string is quite important and so the ZX81

provides a very simple way of finding the length of any string. Try the following program:

```
10 INPUT A$
20 PRINT LEN (A$)
30 GOTO 10
```

You will see that it prints the number of characters that you type in response to the INPUT. In general,

```
LEN('string expression')
```

will give the number of characters in the 'string expression'. (The use of LEN will be discussed in more detail in Chapter 6.)

Arrays

Strings and numbers are the only two types of data that the ZX81 can handle and this is quite sufficient for most purposes. However, the ZX81 does provide a way of using the basic types of data in a more sophisticated way, the *array*.

Consider the problem of reading in five numbers and printing them out on the screen in the reverse order. So far the only method that we could use is:

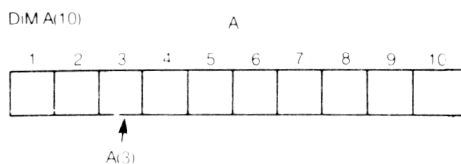
```
10 INPUT A1,A2,A3,A4,A5
20 PRINT A5,A4,A3,A2,A1
```

which is not too bad for five numbers but think what the program would look like if the problem were to reverse a hundred numbers!

What we need to be able to do is to refer to a variable like 'A(I)' where I can take values from 1 to 5 in a FOR loop. Then we could write:

```
20 FOR I=1 TO 5
30 INPUT A(I)
40 NEXT I
50 FOR I=5 TO 1 STEP -1
60 PRINT A(I),
70 NEXT I
```

This is, in fact, exactly what BASIC allows you to do. The collection of variables A(1) to A(5) is called *the array A* and a particular variable A(I) is called an *element* of the array (see Fig. 5.1). The only

*Fig. 5.1.*

complication is that before you can use an array you must tell your ZX81 how many elements the array is going to have. This is done using the DIMension statement:

```
10 DIM A(5)
```

which should be added to the previous program to make it work! If you define a variable as having only five elements and then try to use A(6) you will get an error message for trying to use something that doesn't exist! It is tempting to think that it is better to define arrays larger than you need to try to avoid such error messages but be warned, arrays can quickly use up all the memory that your machine has to offer! There is a similar restriction on the names of arrays as on index variables, i.e. they can only be one letter long! This means that you can only have 26 different arrays called 'A' to 'Z', but this is usually more than enough. Using the DIM statement destroys any arrays that already exist with the same name and creates a brand new array initialised so that each element stores zero.

In addition to being able to define arrays that can be thought of as *rows* of variables (see Fig. 5.1) you can define arrays that correspond to organising variables into tables made up of rows and columns. For example,

```
DIM A (10,10)
```

defines a collection of variables organised into ten rows and ten columns. A particular element of this array can be referred to as A(I,J) where the two indices select the row and column.

The idea of a two-dimensional array can be extended to three-, four-, five- etc. up to 255-dimensional arrays. It is difficult to think of an arrangement of variables that corresponds to the higher dimensional arrays but they are defined and used in roughly the same way as one- and two-dimensional arrays. For example,

```
DIM A(10,20,5)
```

is a three-dimensional array and a typical element is $A(2,1,4)$. There is not much use for arrays with dimensions greater than two. This is fortunate because they tend to use up memory very, very fast!

String arrays

You can form arrays from string variables as well as numbers and this can be used to store and manipulate lists of words. There is one complication, however, in that in some senses a string is already a one-dimensional array of characters. If you define a one-dimensional string array, in fact what you get is a string of fixed length. For example,

```
DIM S$(10)
```

is a string variable that always stores ten characters no matter what (even if some are blank)! To see the difference between the string $R\$$ and the string array $S\$$ try the following program:

```
10 DIM S$(10)
20 LET S$="ABC"
30 LET R$="ABC"
40 PRINT S$;"X";R$;"X"
```

You should see that $S\$$ is ten characters long no matter what you store in it. Notice that for string arrays you can store values in more than one element at a time. You can also store single characters in elements, however. The following lines store Z in S(5)$.

```
50 LET S$(5)="Z"
60 PRINT S$
```

You might be able to see a similarity between this and the slicing notation introduced earlier.

You can handle lists of words by using two-dimensional string arrays. For example, the number reversing program can be used to reverse a list of words:

```
10 DIM A$(5,10)
20 FOR I=1 TO 5
30 INPUT A$(I)
40 NEXT I
50 FOR I=5 TO 1 STEP -1
```

```

60 PRINT A$(I)
70 NEXT I

```

Notice that if you leave out the last index in a two- (or more) dimensional string array this is taken to mean that you want to treat the array as a collection of strings. For example, A\$(5,2) is a single character, but A\$(5) is a string of ten characters. You can even use the slicing notation to pick out substrings. For example, A\$(5,1 TO 2) is a substring starting at character 1 and ending at character 2.

Sinclair BASIC allows very flexible uses of strings and arrays. These can be difficult to understand at first but once you have become accustomed to them you will find that they are very powerful.

A word game

As an example of using string arrays, consider the problem of writing a program to play the game of 'Hangman.' Because the computer cannot 'think up' a list of words for you to guess it is necessary to ask someone else to type in a list for you to guess. Once the list of words has been entered, the player must try to guess each word in turn, letter by letter. Each letter that is entered must be checked against each letter in the word. If it is present then the letter in the word must be replaced by a blank to make sure that the player cannot guess it a second time. When all the letters have been guessed the program moves on to the next word or, if there are none, comes to an end. After this description you should be able to make a good attempt at your own Hangman program before looking at the one below:

```

10 DIM W$(5,10)
20 FOR I=1 TO 5
30 PRINT "WORD = "
40 INPUT W$(I)
50 NEXT I
60 FOR I=1 TO 5
70 LET G=0
80 LET T$=W$(I)
90 LET G=G+1
100 PRINT "GUESS = "
110 INPUT A$
120 LET F=0
130 FOR J=1 TO 10

```

```

140 IF A$(I)<>W$(I,J) THEN GOTO 170
150 PRINT "YES ";A$(I)
160 W$(I,J)=" "
170 IF W$(I,J)<>" " THEN LET F=1
180 NEXT J
190 IF F=1 THEN GOTO 90
200 PRINT "YOU GOT IT IN ";G;" *"
210 PRINT "THE WORD WAS ";T$
220 NEXT I
230 PRINT "GAME OVER"

```

Line 10 defines the array W\$ so that it can hold five words of up to ten letters each. Lines 20 to 50 are used to input the words. The FOR loop starting at line 60 and ending at line 220 repeats the guessing part of the program five times, once for each word. The variable T\$ is used in line 80 to store the word until the end of the game so that the array element can be modified by storing blanks in the place of letters that have been guessed. The guess is input in line 110. Each letter in the word is checked against the current guess by the FOR loop at line 130 to line 180. The variable 'F' is used to check that there are still letters left to be guessed.

This game runs on a 1K ZX81 and takes it to its limits. As you improve your own programming skills you are likely to be dissatisfied with programs like this one that lack extra features that make them more 'user-friendly' or more exciting to play. An obvious improvement in the case of Hangman would be the addition of a graphics routine – but you'll have to wait for Chapter 7 in order to learn about the appropriate graphics skills. Also, before you can embellish this program you'll need to expand your ZX81's memory.

Chapter 6

Functions And Subroutines

At this point in learning BASIC you should be in a position to see that the *expression* is the main way that programs *change* data. Without expressions – arithmetic, conditional and string – BASIC would be reduced to moving values from one place to another. It is only by the use of expressions that values can be combined and compared to produce new results. To make expressions even more useful, BASIC provides a large range of operations that can be used to make expressions, in the form of *functions*. BASIC also enables you to use the GOSUB and RETURN statements to group statements together into functional units or *subroutines*.

In the first part of this chapter we look at the general idea of a function and then move on to examine some of the more common functions available to the ZX81 programmer. The sections that deal with particular functions can be read very quickly or even skipped until you need to use them or until they are used in an example. However, don't skip the section on special functions because these are particularly important.

The idea of a function

Before dealing with the way the ZX81 handles functions, it is worth looking at functions in general. You may already be familiar with the idea of a function from mathematics. For example, $\sin(x)$ is a function. However, the idea of a function isn't really anything to do with advanced mathematics. At its most simple, a function is an operation on data that produces a *single* value as its result. For example, finding the larger of two numbers is an operation on data that returns a single value – the maximum of 3 and 42 is 42, the maximum of 2 and 2 is 2 – and 'maximum' is therefore a function.

The ZX81 doesn't have a maximum function but it is nevertheless a useful one to consider as an example because it is easy to understand.

The standard way of writing a function involves writing its name to the left of the values that it is to operate on enclosed in brackets. In the case of finding the maximum of two numbers a sensible name for this function is 'max' and so the previous two examples can be written as

max(3,42)

and

max(2,2)

Following the usual BASIC convention that anywhere that you can use a constant or a variable you can use an expression, the following is also allowed:

max (COUNT+3,TOTAL*20)

The data values that follow the name of the function are called *parameters*. It is possible for functions to have any number of parameters but all of the functions supplied on the ZX81 only have one.

Functions can be used in expressions just as if they were variables or constants. For example,

LET RESULT=max(3.3,4.2)

would evaluate our function 'max' and store the answer (4.2) in 'RESULT'. (Remember that the ZX81 doesn't have a 'max' function so don't try this example.) You can now see why the condition that a function should give only one result is so important. If a function gave more than one result, which one would be stored in the variable 'RESULT' or which one would be used to evaluate the rest of the expression? As we want to use functions in expressions they can only return one answer. Sometimes it is possible to change something that isn't a function into a function by simply choosing one of the possible answers. For example, the ZX81 has a function SQR which gives the square root of a number. Now if you ask for the square root of four the answer is obviously two, because two times two is four, but it is all too easy to forget that minus two is also the square root of four. A minus times a minus is a positive and so $-2*-2$ is 4 (not -4). The objection that the square root operation isn't a function can be avoided by simply deciding that SQR will be a function that returns the positive square root of a number.

The ZX81's functions

The ZX81 has a very wide range of functions and some of them are so specialised that it is better to deal with them in detail in other chapters. However, there is a central core of functions that you would expect to find in any BASIC and these will be explained in this chapter. A full list, with brief explanations, of all the ZX81's functions can be found in Appendix C of the ZX81 Manual.

The core functions can be divided into three groups: the arithmetic functions such as SQR and ABS; the trigonometrical functions such as SIN and COS; and the string functions such as LEN and CHR\$. In addition there are a number of unclassifiable but very important 'one-off' functions such as RND. All of the functions are entered by a single keystroke and the use of brackets around the parameters is optional. The most important thing is to have some idea of what functions are available, so a brief reading of the description of each function listed below is recommended. However, it is difficult to appreciate, let alone remember, the subtler details of the use of a function until you actually *need* to use it! If you want to see the effect of any of the functions that handle numerical input use the following program:

```
10 INPUT X
20 PRINT SIN X
30 GOTO 10
```

but change the PRINT SIN X to the function that you are interested in. When you RUN this program, remember that you have to enter the values you are interested in. For the functions that operate on strings simply use:

```
10 INPUT A$
20 PRINT LEN (A$)
30 GOTO 10
```

Arithmetic functions

ABS – ABSolute value of a number

The absolute value of a number is obtained by ignoring its sign and treating it as positive, i.e. ABS(-2) is 2 and ABS(2) is 2.

EXP – EXPOnential function

EXP(X), is calculated by raising the exponential number, which has the value 2.718281, to the power of X. That is, EXP(X) is the same as e^x ($e = 2.718281$).

It is difficult to explain why this function is so important but it crops up in just about every area of mathematics (see also the function LN). When using EXP it is well worth being aware of the fact that EXP(X) gets very big for even small values of X. The largest integer that EXP(X) can accommodate on the ZX81 is 88. Larger numbers will cause the error code 6, which means 'result too big', to be displayed.

INT – INTeger value of a number

The INT function is probably the simplest and most widely used of all the arithmetic functions. It will remove the fractional part of a number and turn it into a 'whole' number or an 'integer' by rounding it down. For positive numbers this corresponds to 'chopping off' the fractional part, e.g. INT(3.21) is 3 but for negative numbers things are a little more complicated. Rounding a negative number down looks a little strange – for example, INT (-4.7) is -5. But this is simply because $-4 > -5$, i.e. -5 is *smaller* than -4!

LN – Natural logarithm of a number

The natural logarithm of a number is the power that you have to raise e to give the number. Most people are more familiar with a slightly different form of the logarithm. The logarithm that is given in most log tables is in fact the logarithm to the base ten. In other words, it is the number that 10 has to be raised by to give the original number. Some versions of BASIC include the log to the base ten in the form of the LOG function. The ZX81 lacks such a function but this is no great disadvantage because you can obtain the log to the base ten by using $\text{LN}(X)/\text{LN}(10)$. As LN(X) is the number that you have to raise e to, to get X you should be able to see that $\text{EXP}(\text{LN}(X))$ is X.

PI – π

This is a very odd function in that it has no parameters and always returns the same result ($\text{PI}=3.14159265$) but it is very useful. As everyone knows, the area of a circle of radius r is given by πr^2 and this translates to BASIC as:

```
10 LET AREA=PI*R*R
```

SGN – the SiGN of a number

The sign of a number is +1 if the number is positive, -1 if it is negative and 0 if it is zero. For example, $\text{SGN}(-232)$ is -1 and $\text{SGN}(3239)$ is 1.

SQR – the SQure Root of a number

The square root of a number is a result that when multiplied by itself gives the original number, thus:

$$\text{SQR}(X) * \text{SQR}(X) \text{ equals } X$$

Notice that negative numbers do not have square roots because if you multiply any number, even a negative one, by itself you will get a positive number. If you try to take the square root of a negative number you will therefore get error code A.

The trigonometrical functions

The best known examples of functions are probably the trigonometrical or 'trig' functions. It is beyond the scope of this book to go into detail about the theory of trigonometry and anyway the need to use such functions always arises from a very specific problem. There is, however, one use for the trig functions that is important to nearly every computer user interested in graphics, namely drawing circles. This is explained later in Chapter 8. However, it is worth recalling the fundamental definitions of the trig functions and their relationship with triangles. If the sides of a right angled triangle are labelled A,B,C as shown in Fig. 6.1. then the trig functions are $\text{SIN}(x) = C/A$; $\text{COS}(x) = B/A$; and $\text{TAN}(x) = C/B$. If you know an angle and one side of a triangle then you can use these equations to work out the length of the unknown sides.

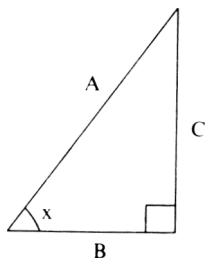


Fig. 6.1.

If you do need to use any of the trig functions, it is important to realise that the ZX81 doesn't measure angles in degrees but in *radians*. If you want to convert an angle in degrees to radians then use:

$$\text{radians} = \text{degrees} * \text{PI} / 180$$

and to convert radians to degrees use:

$$\text{degrees} = \text{radians} * 180 / \text{PI}$$

The three trig functions available on the ZX81 are:

SIN – SINE of an angle measure in radians

COS – COSine of an angle measured in radians

TAN – TANGent of an angle measured in radians

The ZX81 also has available the inverse functions related to these three.

ASN – ArcSiNe

The arcsine of a number is the angle in radians whose SIN is equal to the number, i.e. $X = \text{SIN}(\text{ASN}(X))$. Because SIN gives a result between +1 and -1, ASN(X) can only be worked out for X in this range.

ACS – ArcCoSine

The arccosine of a number is the angle in radians whose COS is equal to the number, i.e. $X = \text{COS}(\text{ACS}(X))$. Because COS gives a result between +1 and -1, ASN(X) can only be worked out for X in this range.

ATN – ArcTaNgent

The arctangent of a number is the angle in radians whose TAN is equal to the number, i.e. $X = \text{TAN}(\text{ATN}(X))$.

String Functions

We have already met some of the string functions in Chapter 5.

CHR\$ – CHaRacter function

CHR\$ n will give the 'character' that is at the nth position in the list of all the ZX81's characters. Notice that as keywords such as LET

count, from the ZX81's point of view, as a single 'character' the function CHR\$ can return a string of more than one letter. CHR\$ will return all the characters that the ZX81 can use even if they cannot be printed on the screen. So, if you type CHR\$(8), or any number up to and including 33, all you will see is a blank screen.

CODE – the code of a character

The CODE function does the opposite to the CHR\$ function in that it returns the position in the list of characters of any particular character. For example, CODE "A" is 38 and CHR\$(38) is "A". If CODE is applied to a string of more than one letter the code of the first character in the string is returned as the result. If the string is null, i.e. contains no characters, the code returned is zero. As CODE expects a string to be input rather than a number you will need to use the following program if you want to try it out. (The same program can be used with LEN and VAL.)

```
10 INPUT A$
20 PRINT CODE (A$)
30 GOTO 10
```

LEN – the LENgth of a string

The LEN function returns the length of any string. For example, LEN "COMPUTER" is eight and LEN "" (the null string) is zero.

STR\$

The STR\$ is a function that is useful for advanced applications. It converts any number (or the result of an expression) to the string of characters that would be displayed if the number (or the result of the expression) were printed. The STR\$ function provides a link between numbers and strings – for example, "JULY"+STR\$(31) works out to the string "JULY 31".

VAL – eVALuate arithmetic expression

The VAL function is the opposite of the STR\$ function in that it converts a string into a number. The string can be any correct arithmetic expression and the resulting number is the value of the expression. For example, VAL "34" is 34 and VAL "3+3*6" is 21. This is an incredibly useful function in that it allows you to enter equations from the keyboard and get your ZX81 to work them out. For example,

```

10 PRINT "WHAT IS YOUR EQUATION? "
20 INPUT A$
30 PRINT "LOWEST X= "
40 INPUT L
50 PRINT "HIGHEST X = "
60 INPUT H
70 PRINT "STEP = "
80 INPUT S
90 FOR X=L TO H STEP S
100 PRINT X, VAL(A$)
110 NEXT X

```

This program will allow you to enter any equation involving X, for example $\text{SIN}(X)+1$ and produce a list of its values for X between L and H. The number of values printed is governed by the value of S which is used as the step size in the FOR loop in line 90. You can imagine that if you add graphics to this program you could draw the graph of the function!

Special functions

There are two functions, RND and INKEY\$, that are so generally useful that it seems worthwhile to treat them on their own and at some length. RND is a function that returns a number so it could otherwise have been treated in the section on arithmetic functions. INKEY\$ returns a character so it could otherwise have been treated as a string function.

RND

RND is a function with no parameters that returns a number in the range 0 to less than 1 which can be treated as if it were random. To say that a computer can give a number at random always sounds like a contradiction and indeed to some extent it is. The point of confusion comes from the use of the word 'random'. If you are using the computer to play a game then all that you need is a sequence of numbers that cannot be predicted by anyone playing the game. In other words, for most purposes a list of numbers can be said to be random if there is no detectable pattern. If you run the following program:

```

10 PRINT RND
20 GO TO 10

```

you should see a list of numbers that shows no obvious pattern. (In fact there is a pattern but it is so complicated it takes a ZX81 to follow it!) This sort of randomness is more correctly called 'pseudo-randomness' and the RND function is a 'pseudo-random number generator'. The numbers that it produces are *evenly spread* throughout the range 0 to less than 1, i.e. any number is equally likely to 'come up' as any other and there should be no discernible pattern that would help someone predict the next number that RND will produce.

The main trouble with RND is that it's not often that we need a random number in the range 0 to less than 1. We normally need the program to do one of a number of different things at random. The best way of doing this is to change the RND into a random whole number between 1 and n where n is the number of anything we want to select from using the formula:

$$\text{INT (RND*N)+1}$$

For example, if you want to program a six-sided dice then you would choose six as the value of N, and

```
10 PRINT INT (RND*6)+1
20 GO TO 10
```

will print numbers 1 to 6 with approximately the same frequency and in such a way that there should be no obvious pattern. The subject of how to use random numbers in programs is too vast to cover in this book but many examples will crop up in other chapters.

The RND function is also special because it is associated with another BASIC command, RAND. The list of numbers that RND produces doesn't go on for ever – eventually, after 65536 values, it repeats itself. Every time you switch the ZX81 on the list starts from the same place. If you want to check this, first switch your ZX81 off and on again and then enter the program that prints a screen full of random numbers. No matter how often you repeat this procedure, remembering to switch off and on again each time, you will get the same numbers in the same sequence. Obviously this is not a good idea if you want to play games because you might eventually learn the sequence that is produced when the ZX81 is first switched on. On the other hand, you might actually want to generate the same sequence each time and if this meant switching the machine off between each run this would also create difficulties. To overcome both these

problems, you can use the RAND command (entered by pressing the key marked RAND to start the sequence off. The command RAND n will start the sequence off from the nth random number in the ZX81's fixed list. For example, if you enter

```
10 RAND 30
20 PRINT RND
30 GOTO 20
```

you will get the same sequence of numbers every time you run it without switching the machine off. However, if you use RAND 0 or RAND without any starting number the ZX81 will use a number that is related to the time that the machine has been switched on to start the sequence. To see this in action try

```
10 RAND 0
20 PRINT RND
30 GO TO 10
```

which prints the first number in the sequence produced by RAND 0 over and over again. You should see that the numbers printed slowly increase, counting the time that the ZX81 has been switched on. The most random sequence that you can produce using RND and RAND can be seen by running

```
10 RAND 0
20 PRINT RND
30 GO TO 20
```

INKEY\$

The function INKEY\$ is closely related to INPUT in that it can be used to 'read in' a single character from the keyboard. The difference is that INPUT A\$ waits for something to be typed on the keyboard until the NEWLINE key is pressed but INKEY\$ doesn't wait. If you try the following program

```
10 INPUT A$
20 IF A$<>" THEN PRINT A$
30 GOTO 10
```

you will have to press NEWLINE before you see anything on the screen. (To stop this program you will need to delete the left-hand quote marks displayed on the bottom line of the screen and then type in STOP as one keypress.) However, if you change the line 10 to

```
10 LET A$=INKEY$
```

the character corresponding to any key that you press appears on the screen at once. (While running this second version of the program try pressing more than one key at a time and try pressing SHIFT and the other keys.) Notice that another difference between INPUT and INKEY\$ is that INKEY\$ doesn't automatically print anything that you type on the bottom of the screen.

Whenever the ZX81 meets the INKEY\$ function it immediately examines the keyboard. If there is a key already pressed the appropriate character is returned by the function. If no key is pressed the function returns the null string. No matter what has happened the INKEY\$ function does *not* wait for a key to be pressed.

The main use of INKEY\$ is in games where the arrow keys used for editing are used to control the movement of something on the screen. For example,

```
10 LET A$=INKEY$
20 IF A$="" THEN GOTO 10
30 IF A$="5" THEN PRINT "LEFT"
40 IF A$="6" THEN PRINT "DOWN"
50 IF A$="7" THEN PRINT "UP"
60 IF A$="8" THEN PRINT "RIGHT"
70 GOTO 10
```

Line 10 gets the character corresponding to any key pressed on the keyboard, if any. Line 20 tests to see if A\$ is the null string, i.e. no key has been pressed, and if it is, sends control back to line 10. Thus the loop formed by line 10 and 20 only stops when a key is pressed. Then lines 20 to 50 test to find out which of the arrow keys are pressed and print an appropriate message. Line 70 repeats the whole program. Notice that if any key other than an arrow key is pressed the loop formed by lines 10 and 20 stops but nothing is printed on the screen. In Chapter 9 an example is given where the same sort of program is used to drive a dot around the screen.

Subroutines GOSUB and RETURN

Functions are one way in which you can increase the range of things that you can write in BASIC. Once you know about functions you are no longer restricted to simple arithmetic! It would be a great advantage if we could define new functions that would make up for

the shortages in ZX BASIC or just to invent new functions that are useful in a particular application. This is indeed possible in most other versions of BASIC. Even ZX BASIC as supplied on the Spectrum has this facility but ZX BASIC on the ZX81 does not. This lack of user-defined functions is not really a major problem with the ZX81's BASIC, it is only an inconvenience.

There is another way to extend the range of things that are easy to do in BASIC and this is to use *subroutines*. Suppose that you wanted to do something like draw a square on the screen in a number of different places. You could work out the program details for drawing a square on the screen and repeat it each time you wanted to draw a square. If the lines of BASIC that define the drawing of a square could be collected together and given a name, then every time you wanted to draw a square you could use the name instead of repeating the list of instructions. So, for example, if the list of instructions to draw a square were collected under the name SQUARE, then using the name SQUARE in a program would draw a square just as if a new command had been added to the BASIC language. This is the idea behind a subroutine. A subroutine is nothing more than a group of BASIC statements that can be used as often as required just by writing their 'name'. The trouble, however, with BASIC subroutines is that they are very limited. You cannot even give a one-letter name to a subroutine. It has to be referred to by the line number of its first line. Also to be really useful it would be an advantage to be able to use parameters in roughly the same way as with functions. So the subroutine SQUARE might be something like SQUARE(X,Y) where the values given to X and Y determined where on the screen the square would be drawn. However, like most versions of BASIC, ZX BASIC makes no provision for parameters of any kind to be used with subroutines! Even with these restrictions the BASIC subroutine is still well worth knowing about and using.

If the lines of BASIC that go to make up the subroutine start at line 'n' then you can use the subroutine by

GOSUB n

Where GOSUB stands for GO to SUBroutine. Indeed, the action of a GOSUB is very like GOTO in that it transfers control to line 'n'. The difference between a GOTO and a GOSUB is that the GOSUB command causes the ZX81 to store the line number of the GOSUB in a special area of memory set aside for the purpose. This stored line number is used by the RETURN statement to transfer control to the

line following the GOSUB when the subroutine has finished. For example, the effect of a GOSUB and a RETURN on the flow of control is:

```

      10 GOSUB 1000 — — — — —
→ 20 PRINT A      |
30 . . .         |
40 rest of program |
                  |
      1000 LET A=56 ← — — — — —
1010 RETURN      |

```

Line 10 transfers control to the subroutine that starts at 1000, which simply stores 56 in the variable 'A'. The RETURN statement at line 1010 ends the subroutine and automatically transfers control back to line 20.

You can use *any* BASIC statement that you can use elsewhere in a program within a subroutine. In particular, there is nothing to stop a subroutine from using GOSUB and transferring control to another subroutine. If you do this then the next RETURN will transfer control back to the statement after the most recent GOSUB. In other words, if one subroutine calls another then RETURN behaves as you would expect it to, by transferring control back to the place that each subroutine was initiated, i.e. a RETURN never forgets where it came from!

This pair of instructions GOSUB and RETURN are all that there are to the BASIC subroutine. All the variables in a subroutine are the same as the variables in the rest of the program; there are no parameters of any kind. As suggested earlier this might lead you to believe that subroutines are not very useful, but this is far from the truth!

Using subroutines

If you read the ZX81 Manual (Chapter 14) you might think that the most useful way to use a subroutine is to replace any piece of program that is needed to print more than once. For example, if in a large program you need the same message over and over again, then it is better to turn that line into a subroutine and GOSUB to it every time it is needed. Although this is an important use of subroutines it is often the case that it is a good idea to form parts of a program into

subroutines even if each part is only used once! The reason for this is that programs that use subroutines are easier to understand, easier to find mistakes in and easier to modify. This is not the sort of statement that anyone can prove because what is easier in this context is clearly a matter of opinion. The use of subroutines in writing BASIC programs will be illustrated by the examples in the rest of the book. If you discover some other method of programming that you like better, then no one will be able to argue with you! All we can say is that a great many programmers agree that subroutines are a good thing and should be used as often as possible! The policy we recommend is to group any BASIC statements together into a subroutine if they are all concerned with achieving a single result or effect.

Chapter 7

Introducing Graphics

One of the main sources of fun in programming any computer is graphics! This chapter starts by introducing the sort of graphics that can be produced using PRINT statements – *low-resolution graphics* – and then goes on to drawing pictures in finer detail – *high-resolution graphics*. You shouldn't be misled into thinking that high-resolution graphics are in some way more useful than low-resolution graphics. They are not more advanced, just different and it's amazing how often a program works better and is easier to write using low-resolution graphics.

Controlling PRINT

So far we have used the PRINT statement to print numbers and strings on the screen either one to a line or next to each other on the same line. Although this is sufficient for many programs there is often a need to control exactly where something will be printed on the screen. In Chapter 3 the PRINT statement was defined as:

PRINT 'print list'

where 'print list' was explained to be a list of items, each one separated by semicolons. The need for the semicolons is simply to show where one item ends and another begins. For example, PRINT TOTAL SUM will try to print a single variable called 'TOTALSUM' (remember blanks are ignored in variable names), while PRINT TOTAL;SUM will try to print two variables 'TOTAL' and 'SUM' next to each other on a single line.

The ZX81 actually allows the use of two symbols to separate print items and each has a different effect on the layout. The semicolon ';'

that we have been using since Chapter 3 simply means 'print the next item without leaving any space'. Using a comma ',' as a separator means move to the next print 'zone' before printing the next item. The ZX81's screen is divided into two print zones – the first 16 printing positions on a line and the last 16 positions on a line. So PRINT "A","B" will print 'A' in column one and 'B' in column 17. You can place more than one separator between any two print items without any ill effects, e.g. PRINT "A",,"B" will cause the ZX81 to move on two print zones before printing 'B'. If you think about it, this means that 'A' will be printed at the beginning of a line and 'B' will appear at the beginning of the next line. There is one special case that is worth commenting on. If either of the separators is placed at the end of a list of print items the automatic starting of a new line is suppressed. For example:

```
1Ø PRINT "A",
2Ø PRINT "B"
```

is the same as

```
1Ø PRINT "A","B"
```

and

```
1Ø PRINT "A";
2Ø PRINT "B"
```

is the same as

```
1Ø PRINT "A";"B"
```

PRINT functions — TAB and AT

The use of different 'print list' separators has certainly increased our control over how things are printed on the screen but we still cannot easily make a print item start at a particular column on a particular line. To achieve this we need something more than different separators. Two special functions, TAB and AT, are provided to control exactly the place where any items start printing. They are 'special' in the sense that although TAB and AT are written like functions, they produce no value as a result of their use and they can be used only as part of a 'print list'. TAB can be used to control the horizontal position on the current line and AT is an entirely general function that can produce output anywhere on the screen.

The general form of the TAB function is

TAB 'arithmetic expression'

and its effect is to move the printing position on to the column given by the value of 'arithmetic expression'. For example,

```
PRINT "A";TAB 10;"B"
```

will print 'A' at column one and 'B' at column 10. You can have as many TABs in a PRINT statement as you need, so

```
PRINT "A";TAB 10;"B";TAB 20;"C"
```

will print 'A' at column 1, 'B' at column 10 and 'C' at column 20.

There are two possible problems that you can arise when using TAB. What happens if you have already gone beyond the column specified by a TAB and what happens if you specify a column that doesn't exist like TAB 35 (there are only 32 columns!)? The answer to the first question is that the ZX81 will move to the correct column *on the next line*. You can see this if you try the following:

```
10 PRINT "A";TAB 15;"B";TAB 10;"C"
```

which prints 'A' at column 1, 'B' at column 15 and 'C' at column 10 on the next line. The answer to the second question is that the ZX81 subtracts 32 from the column number that you specify until it gets a number in the range 0 to 32. For example, TAB 46 has the same effect as TAB 14, because $46-32$ is 14, and TAB 126 has the same effect as TAB 30, because $126-32$ is 94, $94-32$ is 62 and $62-32$ is 30!

The most powerful print function is AT because it can position output at both a particular line and a particular column. The general form of AT is:

AT 'arithmetic expression 1', 'arithmetic expression 2'

The value of 'arithmetic expression 1' is the line number that the next item will be printed on and the value of 'arithmetic expression 2' is the column number. (If the line or column specified is off the screen then you will get an error code B.) Unlike TAB, which numbers printing positions 1 to 32, the columns are numbered starting with 0 and ending with column 31. Similarly, the line numbering goes from 0 (for the top line) to 21 (for the bottom line). For example,

```
PRINT AT 3,5;"A"
```


will print 'A' on line 3 (i.e. the fourth line down) and column 5 (i.e. the sixth printing position). (This might seem a little difficult if you are used to x,y co-ordinates where x is the horizontal distance and y is the vertical distance, because to move to printing position x,y you have to use PRINT AT y,x.)

An interesting point about both TAB and AT is the way that they can be used to move the current printing position and print something without affecting anything that is already on the screen. To see this try the following:

```
10 PRINT "*****"
20 PRINT AT 0,0;
30 PRINT TAB (10);"A"
```

It doesn't matter exactly how many asterisks you use in line 10, they are only there to show the effect of TAB. There are two points to notice about this demonstration. First, you can use AT to move the current printing position anywhere on the screen without actually printing anything, and secondly, line 30 shows that the TAB to column 10 doesn't erase the asterisks from the beginning of the line to column 10. If you want to see that the effect of AT is just the same, substitute

```
30 PRINT AT 0,10;"A"
```

for line 30.

You can have as many ATs in a 'print list' as you desire. Each one moves the printing position to the place indicated before going on to the next item. Not only can you have more than one AT, you can mix AT, TAB and all the other print control symbols in a print list in any way that makes sense to you. For example,

```
10 PRINT AT 10,5;"A";TAB(5);"B";AT 3,10;"C"
```

is accepted without qualms by your ZX81.

```

* * * * *
*           *
*           *
*           *
*           *
*           *
*           *
*           *
*           *
* * * * *
```

Fig. 7.1. Square of asterisks.

As an exercise in using AT, try to write a program that draws a square made of asterisks on the screen (Fig. 7.1). In case you have any problems, one of the many possible answers (there is always more than one way of writing a program) is:

```
10 FOR I=0 TO 10
20 PRINT AT 5,I+10;"*"
30 PRINT AT 15,I+10;"*"
40 PRINT AT I+5,10;"*"
50 PRINT AT I+5,20;"*"
60 NEXT I
```

Lines 20 and 30 print the two horizontal rows of asterisks and lines 40 and 50 print the two vertical columns of asterisks.

A full screen — CLS and SCROLL

The ZX81's screen is 32 characters by 22 lines and sooner or later you are bound to use all the space and still want to print yet more information. If you have finished with everything already on the screen you can use the CLS statement (CLS stands for Clear the Screen). This simply wipes everything off the screen and sets the current printing position back to the top left-hand corner of the screen.

If you are printing things on the screen working from top to bottom there will come a time when you reach the bottom line. If you try to print another line the ZX81 will stop and print an error code 5. As we already know, pressing the CONT key solves this problem by clearing the screen and continuing the program from the point that the screen became full. However, this is sometimes an untidy and unsatisfactory way of handling the screen.

An alternative solution is to use the SCROLL command. Following a SCROLL command, the whole screen moves up by one line, the top line is lost and a new clear line appears at the bottom ready for you to print on. This technique of moving the screen up by one line is known as *scrolling*. You can make sure that your ZX81 never stops because of a full screen by always printing on the bottom line and immediately executing a SCROLL command. To see this in operation try, for example:

```
10 PRINT AT 21,0;RND
20 SCROLL
30 GOTO 10
```

Press **BREAK** when you want to stop this program.

Once you have seen the screen scrolling continuously in this way the idea is not difficult to understand. However, the problem of stopping the screen long enough to read still remains! In practice, the best solution is to keep a count of how many lines you have printed and when you have printed 21 lines jump to a subroutine that contains an **INPUT** command. This will cause the program to pause until the **NEWLINE** key is pressed. For example:


```

10 LET COUNT=0
20 PRINT AT 21,0;RND
30 SCROLL
40 LET COUNT=COUNT+1
50 IF COUNT=21 THEN GOSUB 1000
60 GOTO 20

1000 INPUT A$
1010 LET COUNT=0
1020 RETURN

```

The graphics characters

The combination of **PRINT** and **AT** can obviously be used to place a character anywhere on the screen and, as we saw earlier in this chapter (in the program that prints a square), this can be used to produce limited graphics. This ability really only comes into its own when used with the ZX81's range of graphics characters. These can be produced by pressing **SHIFT** and **GRAPHICS** to change the cursor to  and then pressing any other key. You can leave graphics mode by pressing **GRAPHICS** again. While in graphics mode, pressing any letter or number key will result in an inverse (i.e. white on black) form of the usual symbol being displayed. This inverse mode is useful for high-lighting important messages on the screen but it is important not to get carried away because too many messages in inverse make the screen look very messy. Because of the difficulty of reproducing the ZX81's graphics mode characters in print, we have used the convention that any character shown in square brackets must be entered in graphics mode. So **[A]** means press the 'A' key while in graphics mode and, as we have just discovered, this results in an inverse A appearing on the screen. You can use the **CHR\$** function to produce the inverse character set. To see how this works, run the

following program, remembering to press CONT when the screen is full:

```
10 FOR I=139 TO 191
20 PRINT CHR$(I)
30 NEXT I
```

In other words, the inverse character set occupies positions 139 to 191 in the ZX81's super-alphabet (see Chapter 5).

As well as the inverse characters there are 22 true graphics characters. Eight of these are printed in white on the top row of keys alongside the numbers. Six more can be found on the second row of keys, Q to Y and six more on the next row A to H. To produce any of these characters all you have to do is press the appropriate key while in graphics mode but while also holding the SHIFT key down. Once again to avoid the problems of reproducing in print the ZX81's true graphics characters the convention of using square brackets to indicate graphics mode will be used along with the up-arrow '^' symbol to indicate that the shift key has to be pressed. So, for example [^ I] means press shift and I while in graphics mode. You can also produce the graphics characters by using the correct character codes in the CHR\$ function. To see the complete set, try the following program:

```
10 FOR I=1 TO 10
20 PRINT I,CHR$(I)
30 NEXT I
40 FOR I=129 TO 138
50 PRINT I, CHR$(I)
60 NEXT I
```

If you have been keeping count of the number of graphics characters you will have noticed that we are two short. The missing two are very important. The first is a character that we have been using without much thought since the start of this book – the space character. The second is the space character's inverse – a solid black block. The space character is different from all the other graphics characters in that it is entered in normal K or L modes. Similarly the inverse space character is different from the other graphics characters in that it is entered in graphics mode but without pressing the shift key. You might think that because the space and inverse space are entered differently they are not really worth thinking about as graphics characters. However, once you have started to use the other graphics characters you will

often find the need to use the space and inverse space. The final proof that the space and inverse space are graphics characters is to be found in the character codes used to produce them. The full graphics set corresponds to characters \emptyset to $1\emptyset$ and 128 to 138. $\text{CHR}\$(\emptyset)$ is a space and $\text{CHR}\$(128)$ is a solid block. Similarly, if $\text{CHR}\$(X)$ is a graphics character in the first set (i.e. one of the codes \emptyset to $1\emptyset$) then $\text{CHR}\$(X+128)$ is the graphics character formed by swapping the roles of black and white, that is its inverse character. In this sense there are only 11 graphics characters corresponding to codes \emptyset to $1\emptyset$ and their inverses corresponding to 128 to 138. To see this try:

```
1 $\emptyset$  FOR I= $\emptyset$  TO  $1\emptyset$ 
2 $\emptyset$  PRINT CHR$(I),CHR$(I+128)
3 $\emptyset$  NEXT I
```

The property that adding 128 to the code for a graphics character produces its inverse is in fact true for all the ZX81's characters. If you look at the character set listed in Appendix A of the ZX81 Manual you should be able to see that the character set splits into two halves. The first half from \emptyset to 63 is the first set of graphics characters and the normal character set. The second half from 128 is simply the first half repeated but in inverse form. In other words if I is the code of a normal printable character I+128 is the code for its inverse. It is possible to make use of this simple fact in programs of all sorts. For example, the following program will echo back everything that you type but in inverse form:

```
1 $\emptyset$  INPUT A$
2 $\emptyset$  FOR I=1 TO LEN A$
3 $\emptyset$  LET A$(I)=CHR$(CODE(A$(I))+128)
4 $\emptyset$  NEXT I
5 $\emptyset$  PRINT A$
6 $\emptyset$  GOTO 1 $\emptyset$ 
```

This is an interesting little program because it shows so many things in use at the same time. Notice the use of the special form of the string slicer in A(I)$ to pick out each character in A\$. Notice the use of CODE to find the character code of A(I)$ and the use of CHR\$ to convert the modified code back into a character. It is worth checking that you understand this program completely before going on!

Apart from having to be entered in a different mode, the graphics characters behave like any others. For example, they can be used in

PRINT statements and strings. The program that printed a square of asterisks can be changed to print a better-looking square (Fig. 7.2) using graphics characters:

```
10 FOR I=1 TO 9
20 PRINT AT 5,I+10; "[" ^ 6]"
30 PRINT AT 15,I+10; "[" ^ 6]"
40 PRINT AT I+5,10; "[" ^ 8]"
50 PRINT AT I+5,20; "[" ^ 8]"
60 NEXT I
```



Fig. 7.2. Square using graphics characters.

Remember that [[^]6] means press shift and 6 while in graphics mode. The trouble with this square is that the corners are missing and putting them in is a matter of printing the 'L' shaped graphics symbols at the correct positions. This is left for you to remedy.

As you can imagine, drawing more complicated shapes using the graphics characters is very difficult. Fortunately, apart from drawing the occasional 'thick' horizontal or vertical line, the graphics characters are normally used in small numbers to print special shapes. For example, if you want to print the outline of a ship during a game you could use:

```
PRINT AT Y,X; "[" ^ Q][^6][^6]
```

which will print a ship at line Y and column X.

If you would like to see a ship move across the screen try:

```
10 FOR X=0 TO 27
20 PRINT AT 5,X; "[" ^ Q][^6][^6]"
30 NEXT X
```

Notice the space left before the first graphics character in line 20. If you want to know what the space is for try leaving it out!

Another way of using the graphics characters is as shading. If you examine the graphics characters once more you will see that there are



Fig. 7.3. Test-tube.

16 solid black and white characters and six characters made up from small dots – that is the characters that can be seen on the third row of keys. These characters can be used to produce a third tone on the TV screen so that with a little planning you can produce pictures in black, grey and white. For example, the following program draws a ‘test-tube’ like shape (Fig. 7.3) filled with a greyish liquid. Notice that there is a space between the two graphics characters in lines 10 and 20.

```
10 PRINT AT 10,5;“[ ^ 8] [ ^ 5]”
20 PRINT AT 11,5;“[ ^ 8] [ ^ 5]”
30 PRINT AT 12,5;“[ ^ 8][ ^ H][ ^ 5]”
40 PRINT AT 13,5;“[ ^ 8][ ^ H] [ ^ 5]”
50 PRINT AT 14,5;“[ ^ 8][ ^ H][ ^ 5]”
60 PRINT AT 15,5;“[ ^ 8][ ^ H][ ^ 5]”
70 PRINT AT 16,5;“[ ^ 8][ ^ H][ ^ 5]”
80 PRINT AT 17,5;“[ ^ 2][ ^ F][ ^ 1]”
```

In practice, using the shaded characters is a matter of trying things out and seeing what they look like. Using the ZX81’s graphics characters is much like trying to make pictures on a typewriter by typing letters in the right place.

It is important to remember that graphics characters are no different from ordinary characters and it is quite possible to mix them. For example, you can print messages on the screen surrounded by impressive borders:

```
10 LET A$=“[ ^ H][ ^ H][ ^ H][ ^ H][ ^ H][ ^ H]”
20 PRINT AT 10,10;A$
30 PRINT AT 11, 10;“[ ^ H]ZX81[ ^ H]”
40 PRINT AT 12,10;A$
```

Notice the use of the string A\$ to hold the graphics characters to save having to write them out twice! Another reason for mixing graphic characters and ordinary characters is to produce better shapes. For example:

```
1Ø PRINT " [ ^6]"
2Ø PRINT "[ ^2]Ø[ ^7][ ^7]Ø[ ^7]"
```



Fig. 7.4. Racing car graphics.

Could be used as a car-like shape in a racing game (see Fig. 7.4). (Notice that there are two spaces before the graphics character in line 1Ø.)

High-resolution graphics

If you look at the ZX81's graphics characters on the top two rows of keys you will see that they are all made by dividing a character square into four quarters and colouring some of the squares white and some black. If you imagine the whole of the ZX81's screen with each character position divided into four in this way you should be able to see that by picking the correct graphics character you can set any of the small squares to black or white. If you had enough patience you could draw lines and shapes by picking the correct combination of graphics characters on a grid of squares twice as fine as the normal character screen. In other words, you can pick any of 64 positions horizontally and any of 44 positions vertically. To make this easier, ZX BASIC has two extra commands PLOT and UNPLOT. PLOT can be used to set any of the small squares to black and UNPLOT can be used to set any of the small squares to white. This is how high-resolution graphics are produced on the ZX81. There is nothing new here, the low-resolution graphics characters that were introduced earlier are simply used to produce the finer detail. Both PLOT and UNPLOT will select the correct graphics character to print at a character position so that only the single point that you have specified is changed.

The only complication is that the position of the high-resolution point is specified in a slightly different way to the PRINT AT command. The horizontal position is easy; the first column is on the left-hand side and is numbered zero. However, the vertical position is

the opposite way round to the numbering used for the print lines. While the first printing line is at the top of the screen, the first plotting line is at the bottom of the screen. Once again the numbering starts from zero so the bottom left-hand corner is column 0, row 0. The column number is usually referred to as the x co-ordinate and the row number is the y co-ordinate. The form of the PLOT and UNPLOT commands is:

PLOT 'x co-ordinate', 'y co-ordinate'

and

UNPLOT 'x co-ordinate', 'y co-ordinate'

As with all BASIC commands, you can use full arithmetic expressions in place of 'x co-ordinate' or 'y co-ordinate'. So PLOT 0,0 produces a black dot in the bottom left-hand corner and PLOT 63,43 produces a black dot in the top right-hand corner. Notice that if the dot is already black at that position specified in a PLOT then nothing changes. The same is true of UNPLOT if the dot at the position is already white.

This is all there is to know about high-resolution graphics! However, using this knowledge to good effect is another matter. You will find plenty of examples in the next chapter but it is worth using the following program to check that you understand the high- and low-resolution co-ordinates.

```
10 PRINT AT 0,0;"HIGH OR LOW ?"
20 INPUT A$
30 INPUT X
40 INPUT Y
50 IF A$(1)="H" THEN PLOT X,Y
60 IF A$(1)="L" THEN PRINT AT Y,X;"[ ]";
70 GOTO 10
```

Try entering both high- and low-resolution co-ordinates and see if you are right about where the point appears on the screen. Notice that if you are using a 1K ZX81 then you might find that this program ends with an error code 5 at some point. This is simply because you don't have enough memory to display a full screen of points – there is no error in the program! Be careful not to specify points that are off the screen or the program will end with yet another error code.

Finally, to bring this chapter to a close, here are two versions of a

simple program. The first uses low-resolution graphics:

```
10 LET X=INT(RND*32)
20 LET Y=INT(RND*22)
30 PRINT AT Y,X;"[ ]";
40 GOTO 10
```

the second uses high-resolution graphics:

```
10 LET X=INT(RND*64)
20 LET Y=INT(RND*44)
30 PLOT X,Y
40 GOTO 10
```

Try both programs and see what the effect looks like in both resolutions. Notice that if you only have a 1K ZX81 then at some point both programs will end with an error code 5, indicating that there is not enough room on the screen. Graphics take a lot of memory!

Chapter 8

Using Graphics

The last chapter introduced all of the elements needed to write programs that make good use of the graphics facilities of the ZX81. However, as with every aspect of programming, knowing the commands isn't the same as knowing what to do with them! The purpose of this chapter is to put some of the information introduced in the last, and earlier chapters, into use and so show how things can be done in practice. Although all of the examples are about some aspect of graphics this is not to say that the only thing to be learned from them is how to use graphics.

Drawing

One of the main difficulties in using high-resolution graphics is in going from the fundamental command `PLOT X,Y` which produces a single dot to more complicated shapes such as lines and circles. Many versions of BASIC avoid this problem by introducing statements such as `LINE` to draw a line and `CIRCLE` to draw a circle, but ZX BASIC on the ZX81 lacks any such commands. This is only a slight problem because it is not difficult to learn how to draw these elementary shapes using only `PLOT`.

The first problem to consider is how to draw a straight line between two points. For example, how do you draw a line between the points $10, 10$ and $20, 20$? Using some simple algebra that really needn't worry us, it is possible to find the equation of the line that connects two points whose co-ordinates are X_1, Y_1 and X_2, Y_2 :

$$Y = M * X + C$$

where

$$M = (Y_2 - Y_1) / (X_2 - X_1)$$

and

$$C=(Y1*X2-Y2*X1)/(X2-X1)$$

These equations may look difficult at first sight but with the help of the ZX81 they can easily be worked out. For any two points X1,Y1 and X2,Y2 you first have to work out M and C and then use $Y=M*X+C$ to work out the values of Y for all the X values that are on the line. Which X values are on the line? Obviously all the values between X1 and X2. To see this in action try the following short program:

```

10 INPUT X1
20 INPUT Y1
30 INPUT X2
40 INPUT Y2
50 PLOT X1, Y1
60 PLOT X2, Y2
70 INPUT A$
80 LET M=(Y2-Y1)/(X2-X1)
90 LET C=(Y1*X2-Y2*X1)/(X2-X1)
100 FOR X=X1 TO X2
110 PLOT X,M*X+C
120 NEXT X
130 GOTO 10

```

Lines 10 to 40 get the co-ordinate values of the two points X1,Y1 and X2,Y2. The next two lines 50 and 60 plot the points so that you can see where they are. To move on to the second half of the program, the section that plots a line between the two points, you have to press NEWLINE because of the INPUT at line 70. Lines 80 and 90 calculate M and C ready to plot the line. Finally, the line is plotted by the FOR loop included in lines 100 to 120. Notice that the program will only work as long as X1 is smaller than X2 because of the way the FOR loop is set up. As an exercise try to change the program so that it will draw a line between any two points – it's not difficult and only involves adding a single IF statement and a STEP to the FOR statement.

After the straight line, the next most widely used shape is probably the circle. Once again, to draw a circle you need to have an equation that gives points that lie on the circle. In fact, this time it is easier to

use two equations – one that gives the X co-ordinate and one that gives the Y co-ordinate:

$$X=R*\text{COS } T + X1$$

and

$$Y=R*\text{SIN } T + Y1$$

These equations may look a little difficult because they involve the functions SIN and COS but the ZX81 finds them very easy to work out! In the equations, R is the radius of the circle and X1,Y1 is the point at its centre. The only quantity that remains to be explained is the variable T. If you calculate the two equations for various values of T then you will always produce points that lie on the circle. In fact, for *any* value of T you will get points that lie on the circle. Try the following program:

```
10 INPUT T
20 PLOT 10*COS(T)+30,10*SIN(T)+20
30 GOTO 10
```

As you enter different values of T you will see points appearing in the shape of a circle. By comparing line 20 with the equations you should be able to see that the circle has a radius of 10 and is centered on the point 30,20 which is roughly in the middle of the screen. It just so happens, for a mathematical reason that need not concern us, that values of T between 0 and 6.28 produce all the points that lie on the circle. The magic number 6.28 is, in fact, for the same mathematical reason, twice the value of the constant PI. So, we can draw a circle (Fig. 8.1) by using:

```
10 FOR T=0 TO 2*PI STEP .1
20 PLOT 10*COS(T)+30,10*SIN(T)+20
30 NEXT T
```

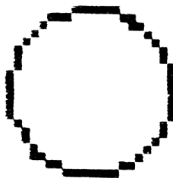


Fig. 8.1. A circle.

There are equations for other shapes but in general most shapes can be made up from lines and circles.

Sketching

In the chapter on functions a program was given that used the INKEY\$ function to detect which of the 'arrow' keys was being pressed. This program can be used to produce a drawing or sketching program:

```

10 LET X=30
20 LET Y=20
30 LET DRAW=1
40 LET A$=INKEY$
50 IF A$="5" THEN LET X=X-1
60 IF A$="6" THEN LET Y=Y-1
70 IF A$="7" THEN LET Y=Y+1
80 IF A$="8" THEN LET X=X+1
90 IF A$="D" THEN LET DRAW=DRAW*-1
100 IF DRAW=1 THEN PLOT X,Y
110 GOTO 40

```

The workings of the program are essentially the same as the one given in Chapter 6. Depending on which key is pressed, the X or Y co-ordinate is updated and the new point plotted by line 100. The only extra detail is that the variable DRAW is used to decide if the point should be plotted or not. If the D key is pressed the value of DRAW is multiplied by -1. As DRAW starts off with a value of 1, multiplying it by -1 changes it to -1. Multiplying it a second time by -1 changes it back to 1 and so on. Each time the 'D' key is pressed the value in DRAW changes its sign. The IF statement in line 100 simply checks to see if DRAW is 1 or -1. If it is 1 then the point is plotted; otherwise only the co-ordinates are updated.

A definite improvement to the program is to add the ability to rub out or erase lines that have already been drawn. This is not difficult as all you have to do is add another variable like DRAW and use it to choose between PLOT and UNPLOT commands. The extra lines of program and the change to line 100 that are necessary to incorporate this feature are given below but before you look at them see if you can work it out for yourself.

```

35 LET ERASE=1
95 IF A$="E" THEN LET ERASE=ERASE*-1
100 IF DRAW=-1 THEN GOTO 40
105 IF ERASE=-1 THEN UNPLOT X,Y
106 IF ERASE=1 THEN PLOT X,Y

```

Notice that these lines of BASIC have to be *added* to the previous program. If you are using a 1K ZX81 it is possible to draw so many lines on the screen that you run out of memory. Also, there are no checks to see if the point is about to go off the screen so use the program with care or, if you have a 16K machine, add a routine to detect the edges of the screen and prevent the point from going beyond them.

Graph plotting

Another program that was given in the chapter on functions was one to work out the values of an equation using the VAL function. It is much more useful to be able to see the 'shape' that an equation produces and the ZX81 can be used to produce rough graphs of any equation. The idea is very simple; just work out the values of the equation over the range specified and plot points corresponding to each X and Y value. The only trouble is how do you make sure that the X and Y values of the function fall into the permitted 0 to 63 and 0 to 43? The answer is that both the X and Y values have to be changed – they have to be 'scaled' to fit the screen. The scaling for the X values is easy because we know both the maximum and minimum values beforehand. If the maximum X value is XMAX and the minimum X value is XMIN, then to scale the X values to fall in the range 0 to 63 we use:

$$XS = \frac{63 * (X - XMIN)}{(XMAX - XMIN)}$$

The rationale behind this equation is not difficult to understand. If you subtract XMIN from X it ranges between 0 and XMAX-XMIN. If you then divide by (XMAX-XMIN) the result ranges between 0 and 1. Finally, multiplying by 63 gives a quantity that ranges between 0 and 63 which is what is actually required. The same reasoning can be used to scale the Y values but we don't know the values of YMAX and YMIN. The only way that these values can be found is first to

calculate the equation for each value of X and see what the largest and smallest values of Y are. The only other question is how many values of X is the equation worth calculating for? The answer is that each value of X should differ by an amount that moves the plotted point on by one screen position. So, the X step size should be:

$$S=(XMAX-XMIN)/63$$

After this very long discussion, it is now time for the program:

```

10 INPUT A$
20 INPUT XMAX
30 INPUT XMIN
40 LET X=XMIN
50 LET YMIN=VAL A$
60 LET YMAX=YMIN
70 LET S=(XMAX-XMIN)/63

80 FOR X=XMIN TO XMAX STEP S
90 LET Y=VAL A$
100 IF Y>YMAX THEN LET YMAX=Y
110 IF Y<YMIN THEN LET YMIN=Y
120 NEXT X

130 FOR X=XMIN TO XMAX STEP S
140 LET Y=VAL A$
150 PLOT (X-XMIN)/(XMAX-XMIN)*63,
      (Y-YMIN)/(YMAX-YMIN)*43
160 NEXT X

```

The program will run on a 1K ZX81 but you may find that sometimes you run out of memory before the graph is completed. This problem can only be overcome by plugging in extra RAM *before* you start to type in the program.

There are three parts to the program. Lines 10 to 70 set up the variables and constants needed later in the program. Lines 80 to 120 find the values of YMAX and YMIN. Finally, lines 130 to 160 actually plot the graph of the equation. The only unexplained part of the program is the section that finds YMAX and YMIN. This is not difficult to understand. At the start YMAX and YMIN are both set to the same value, the value of equation when X equals XMIN. This is achieved by lines 40, 50 and 60. The equation is then worked out for each value of X that will be plotted and the value of Y is compared

with the current values of YMAX and YMIN (by lines 1000 and 1100). Obviously, if Y is larger than the current value of YMAX it should be used to replace the current value. In other words, YMAX is the largest value of Y that 'we have seen so far' and after we have seen all the values it becomes the largest of all values. The same reasoning holds for YMIN but it is only changed if Y is smaller than the current value. In other words, YMIN is the smallest value we have seen so far.

When using this program it is important to keep a few things in mind. First, you can type in any equation involving X such as X^2 or $\sin X$ but the more complicated the equation the longer the program will take to work it out. Secondly, you can specify any values for XMAX and XMIN and the graph will still fit on the screen but you might not see a very interesting curve. Finally, the middle part of the program takes rather a long time to find YMAX and YMIN – so be patient. A method of speeding this program up will be discussed in the next section so either save the program on tape or leave it in your ZX81 until you have finished reading the next section. To demonstrate the program, you might try the following equations:

X^2	XMAX=10	XMIN=-10
$X^3 - 25X$	XMAX=8	XMIN=-8
$\sin X$	XMAX=10	XMIN=0
$\sin X + \sin(2X)$	XMAX=10	XMIN=0
$\arctan X$	XMAX=10	XMIN=0

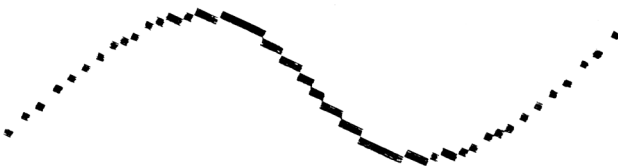


Fig. 8.2. $X^3 - 25X$.

FAST, SLOW and PAUSE

There are three ZX BASIC commands that are closely connected with graphics that haven't been introduced yet – FAST, SLOW and PAUSE. Normally the ZX81 spends part of its time running your program and part of its time constructing the screen display. In fact, the ZX81 regards the work it has to do to display the screen as rather



*Fig. 8.3. SINX + SIN(2*X).*

more important than running your program! BASIC programs are only obeyed in the brief time between each TV frame! This obviously slows your programs down. If you want to run a program at top speed then there is something that can be done. You can instruct the ZX81 to ignore the screen display and devote itself entirely to your program by using the FAST command. However, following FAST you will only be able to see the screen display when the ZX81 isn't running a BASIC program. That is, for example, when it is waiting for you to type something on the keyboard. To change back to the normal operation all you have to do is use the SLOW command. You can use SLOW and FAST within programs to switch off the screen display during parts of the program that do a lot of calculation and then switch it back on again when there is something to be seen. For example, in the graph plotting program given in the last section, after the INPUT commands there is a lot of calculation to be done but nothing to see until the FOR loop at line 130. The program can be speeded up by adding:

```
35 FAST
```

and

```
125 SLOW
```

You will see that the display vanishes after you enter the value of XMIN and then reappears when the graph is produced. If you didn't want to see each point plotted then you could move the SLOW

command to the very end of the program and only show the finished graph. Careful use of FAST and SLOW can result in programs running faster without any loss of useful information. If you have enough memory (that is, with the 16K ZX81) then it is a neat idea to show a message on the screen to the effect that the display is about to vanish while the ZX81 gets on with its calculations.

The ZX BASIC command PAUSE can be used for many things but one of its uses is in allowing a user to view the screen for a given amount of time. This is useful in SLOW mode and it is particularly useful in FAST mode. The command, PAUSE N will make the ZX81 do nothing except display the screen for N fiftieths of a second. For example, PAUSE 100 will make the ZX81 pause for 100 fiftieths of a second or, in other words, two seconds. The time that the ZX81 pauses for can be cut short by pressing any key. So, if you get bored waiting for the two seconds following PAUSE 100, all you have to do is press a key and your ZX81 will get on with the rest of the program. One last detail concerning the PAUSE command that is worth knowing is that a pause longer than 32767 fiftieths of a second will not be timed and so it will go on for ever or until you press a key. For example, PAUSE 40000 will make your ZX81 wait and display the screen until you press a key.

Animation

The basic idea that lies behind making shapes move on the screen is very simple. All you have to do is PRINT or PLOT the shape at its current position. To move it to a new position, first PRINT enough blanks or use enough UNPLOT commands to remove it from its old position and then PRINT or PLOT it at its new position. This sequence of displaying the shape, removing the shape and then displaying it at a new position is all there is to moving graphics. There are some practical problems with achieving smooth movement, however, because smooth flicker-free movement depends on how fast you can repeat the cycle. To see this in action try the simple program:

```
10 PRINT AT 5,0;">"
20 FOR X=0 TO 30
30 PRINT AT 5,X;" "
40 PRINT AT 5,X+1;">"
50 NEXT X
```

```

60 PRINT AT 5,31; " "
70 GOTO 10

```

which appears to make an arrow head fly from the left to right on the screen. Line 10 prints the arrow at the first position and then line 30 removes it by printing a blank. Line 40 prints it at its new position one place further to the right and this loop continues until the far right of the screen is reached. You might like to write a similar program that moves a dot from left to right on the screen using PLOT and UNPLOT.

There is another way of making sure that the old shape is blanked out. If the shape only moves by one position at a time then it is sometimes possible to make it 'self-blanking' by including a trailing blank. An example of this technique had already been given in the previous chapter where a ship was made to move across the screen. The following program, however, uses the technique as the basis for a racing game (see Fig. 8.4).

```

10 GOSUB 500
20 FOR Y=3 TO 9 STEP 3
30 GOSUB 1000
40 NEXT Y
50 DIM X(3)
60 LET I=INT(RND*3)+1
70 LET X(I)=X(I)+1
80 LET X=X(I)
90 LET Y=I*3
100 GOSUB 1000
110 GOTO 60

500 FOR Y=21 TO 39
510 PLOT 62,Y
520 NEXT Y
530 LET X=0
540 RETURN

1000 PRINT AT Y,X,;"  [ ^ 6]"
1010 PRINT AT Y+1,X;"  [ ^ 2]0[ ^ 7][ ^ 7]0[ ^ 7]"
1020 IF X<24 THEN RETURN
1030 PRINT AT 5,0;"NUMBER  ";1;" WINS"

```

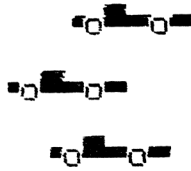


Fig. 8.4. Racing game.

Subroutine 1000 will print the racing car graphic, also introduced in Chapter 7, at line Y and column X. To make the racing car self-blanking an extra space character has to be included in each PRINT statement. (So there are three spaces before the graphics character in line 1000 and one space before the first graphics character in 1010.) The last part of the subroutine checks to see if the racing car just printed has passed the finishing line, that is, has an x co-ordinate greater than 23. If it has then the number of the car is printed by line 1030. The first part of the program, lines 10 to 40, uses subroutine 1000 to print three cars on different lines. It also calls subroutine 500 which plots the finishing line. The second part of the program moves cars at random. Each car's horizontal position is stored in an element of the array X – for example, the first car's position is in X(1). By generating the index I at random the x co-ordinate of a random car has one added to it. Once again, subroutine 1000 is used to print the car at its new position.

There are so many improvements that can be made to this program that it is almost better to think of it as the starting point for a game. However, any improvements will need a 16K ZX81 as the program only just fits into a 1K ZX81 as it stands. The sort of improvement that you might like to try is to add a betting routine so that you have a sum of money to bet on the winner and the game is repeated until you are broke.

Target practice

The final program in this chapter is a small target practice game. No new methods are used to implement this game but it does serve to bring most of the methods introduced so far together in one program (or rather, two versions of one program!). The first is a 1K version and the second is a slightly improved 16K version. The elements of the game are quite simple. A dollar sign is printed at the bottom of the

screen and can be moved left and right using the arrow keys. At the same time an asterisk moves from left to right at random at the top of the screen. The object of the game is to move the dollar to be directly under the asterisk and then fire at it by pressing the 'F' key. It would be nice to show a bullet or missile moving up the screen from the dollar to the asterisk and then show an explosion if it was a hit, but you can do only so much in 1K of memory. Before looking at the program given below try to write your own – you should know enough by now.

```

10 LET X=15
20 LET XT=INT(RND*32)
30 LET AS=INKEY$
40 IF AS="5" THEN LET X=X-1
50 IF AS="8" THEN LET X=X+1
60 IF AS="F" THEN GOSUB 1000
70 PRINT AT 21,X;" $ "
80 PRINT AT 1,INT XT;" *"
90 LET XT=XT+RND
100 IF XT>30 THEN LET XT=0
110 PRINT AT 1,INT XT;"*"
120 GOTO 30

1000 IF INT XT<>X+1 THEN RETURN
1010 PRINT AT 10,0;"HIT"
1020 PAUSE 50
1030 PRINT AT 10,0;" "
1040 RETURN

```

The horizontal position of the dollar sign is stored in X and the horizontal position of the asterisk is stored in XT. The dollar is moved from side to side by the usual method of adding or subtracting from the x co-ordinate according to which key arrow key is pressed. Notice that the dollar is self-blanking because of the blanks on either side of it. (Once again if you want to see what the blanks are doing, try leaving them out!) The position of the asterisk is altered by adding a random quantity to it at line 90. It is blanked by line 80 and printed at its new position by line 110. If it happens to reach the far right of the screen before it is hit then its horizontal position is set to zero and it starts moving across the screen again (line 100). Subroutine 1000 is called if the F key is pressed. All it does is to compare the two horizontal co-ordinates and if they are the same it prints the word HIT on the screen.

If you have a 16K ZX81 there is plenty of scope for adding to this program. For a start, you might like to change subroutine 1000 to show a missile moving toward the asterisk and an explosion when the asterisk is hit. If you replace the previous subroutine 1000 with the one given below then it is exactly what happens:

```

1000 FOR Y=20 TO 1 STEP-1
1010 PRINT AT Y,X+1;":"
1020 PRINT AT Y,X+1;" "
1030 NEXT Y
1040 IF INT XT<>X+1 THEN RETURN
1050 FOR I=1 TO 10
1060 PRINT AT 1,INT XT; "[" ^ 8]"
1070 PRINT AT 1,INT XT;" "
1080 NEXT I
1090 STOP

```

The first part of the subroutine prints a colon and then blanks it out to give the impression of something moving up the screen toward the target. Once the colon reaches the top of the screen a check is made by line 1040 to see if the asterisk has been hit. If it has then lines 1050 to 1080 produce an explosion effect by printing [^ 8] and blanking it out 10 times in the same place. This makes the graphics character appear to twinkle and flash.

There are many more things that you could do to improve any of the short programs given in this chapter. After all, they are only supposed to be examples of how to do things. If they were complete and finished programs there would be a lot of BASIC in each that would obscure the point. Use these programs as starting points for your own experiments and then go on and invent ideas and implement your own games or serious programs. There is much to be learned by looking at other people's programs but there is much more to be learned by looking at your own!

Chapter 9

Logic And Other Topics

In this final chapter a number of different topics are introduced. It would be misleading to regard this as a collection of advanced topics just because they have been left until last. Using the BASIC and other information dealt with in earlier chapters you should find it possible to write any program that you want to. However, there are facilities on the ZX81 that, although not entirely necessary, do make things easier. This chapter collects together these extras and explains how they work. The commands include AND, OR, NOT, PEEK, POKE, USR, CLEAR and REM.

Logic and the conditional expression

In everyday speech we often say things like, 'did you buy apples and oranges?' or 'do you prefer tea or coffee?'. The use of words like *and* and *or* are so common that we rarely stop to think about them. It would be a great advantage if the use of *and* and *or* could be extended to BASIC conditional expressions and, indeed, in ZX BASIC you can write expressions such as:

$A < 0$ AND $B = 3$

$A < 0$ OR $B = 3$

The meaning of each of these expressions is in line with the usual English meaning of AND and OR. The first expression evaluates to *true* if both of the conditions ' $A < 0$ ' and ' $B = 3$ ' are true, and the second expression is true if either of the two conditions is true. As well as AND and OR the ZX81 also allows the use of NOT which simply changes the value of a conditional expression from 'true' to 'false' and vice versa. For example, ' $3 = 2$ ' is false but 'NOT $3 = 2$ ' is true.

You can combine AND, OR and NOT conditions to make more

complicated expressions that will evaluate to one of the values 'true' or 'false'. You will recall that 'true' is represented by 1 and 'false' by \emptyset . Conditional expressions that include AND, OR or NOT are usually called logical expressions and we can now rewrite the definition of the IF statement as:

IF 'logical expression' THEN 'BASIC statement'

For example, if you want to check that you are not using an X co-ordinate that goes outside the screen area then you could use:

IF $X < \emptyset$ OR $X > 63$ THEN ...

in place of the two IF statements that would be required without the use of OR.

Forming logical expressions to test for *overall* conditions is usually straightforward. However, there are a few traps that even experts fall into. If you want to translate the English statement, 'a equals b and c' then you must repeat the condition '='. That is, you must use:

$A=B$ AND $A=C$

and not

$A=B$ AND C

which will give a result that depends on whether C is \emptyset or 1. You should also be careful when using NOT. For example,

NOT($A=B$ AND $A=C$)

isn't the same as

NOT($A=B$) AND NOT ($A=C$)

To see that this is the case try working the two expressions out for a few values of 'A', 'B' and 'C'. The moral is that you should always beware of using logical expressions without thinking about *exactly* what they mean.

To close the subject of logical expressions it is worth introducing the idea of a *truth table*. If you consider the logical expression:

A AND B

where 'A' and 'B' are variables that are either \emptyset for false or 1 for true. You can draw up a table that lists the result of the expression for all possible values of 'A' and 'B', thus:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Such a table is called a truth table because it lists the conditions under which the logical expression is true or false. You can draw up truth tables for any logical expression and this is one way to check that you understand what is happening. For example, OR and NOT have the following truth tables:

A	B	A OR B	NOT A
0	0	0	1
0	1	1	1
1	0	1	0
1	1	1	0

The OR that we have used so far is not entirely equivalent to the English word 'or'. Most uses of the English 'or' mean 'one or the other but not both'. For example, 'You can have jam or marmalade' means that you can pick one but not (normally) both! However, the logical OR means that you can have either one or both (look at the truth table if you are unsure of this). The logical OR is more properly called the *inclusive or* because it includes the possibility of both. The usual English 'or' is known as the *exclusive or* because it excludes the possibility of both. You can make up a logical expression that is equivalent to the exclusive or:

exclusive or = (NOT(A) AND B) OR (A AND NOT(B))

as can be seen from the truth table:

A	B	(NOT(A) AND B) OR (A AND NOT(B))
0	0	0
0	1	1
1	0	1
1	1	0

Logic crops up in some strange places.

Mixed expressions

The idea that expressions were the fundamental way of getting things done in BASIC was introduced very near the start of this book. ZX BASIC allows you to use three different types of expression – arithmetic, logical and string. In principle, each of these types involve quantities that cannot really be mixed up with one another. For example, it doesn't make sense to try to mix numbers and strings in a single expression. In general the result of an expression is the same type as the quantities that make it up. That is, an arithmetic expression evaluates to a number, a logical expression evaluates to a logical value, and a string expression evaluates to a string! This is very simple and for the most part is sufficient. However, there are some tricky ways of forming mixed expressions in ZX BASIC. These are sometimes useful but it should be kept in mind that other versions of BASIC do things differently – mixed expressions are definitely non-standard.

The most obvious form of mixed expression is to use logical expressions within arithmetic. For example, $(2 > 3) + (4 > 3) + (4 = 4)$ evaluates to 2 because the first bracket is 'false' and 'false' is represented by 0; the contents of the second and third brackets are 'true' and 'true' is represented by 1. As an example of a useful mixing of logic and arithmetic, consider the problem of working out tax at two different rates – 30% and 70%, say. You could write a pair of IF statements to decide which calculation should be carried out, for example:

```
IF R$="LOW" THEN TAX=30*MONEY/100
IF R$="HIGH" THEN TAX=70*MONEY/100
```

where R\$ is a string that is used to indicate the tax rate in an obvious

way. However, it is possible to combine both of these possible calculations into one mixed expression:

$$\text{TAX} = (\text{R\$} = \text{"LOW"}) * 4\emptyset * \text{MONEY} / 1\emptyset\emptyset + (\text{R\$} = \text{"HIGH"}) * 7\emptyset * \text{MONEY} / 1\emptyset\emptyset$$

This expression may look a little strange but its interpretation is very simple. If R\$ is not equal to 'LOW' then the first bracket works out to 1 and the second works out to \emptyset . You should be able to see that this selects the correct calculation out of the entire expression – because multiplying by one has no effect but multiplying anything by zero gives zero. You can use this sort of short-cut to simplify many programs and squeeze them into 1K. However, be warned that if you become too dependent on this method you will find it difficult to switch to any other version of BASIC.

The idea of a truth table was introduced in the last section and it is possible to use mixed expressions to get your ZX81 to do all the work involved in constructing *any* truth table. By using the fact that VAL will evaluate an expression that includes logical operations and that the logical values true and false are 1 and \emptyset , you should be able to write a program to evaluate a logical expression for all possible values. For example, if you are interested in seeing the truth table for a logical expression involving two conditions, called A and B for simplicity, then try the following program:

```
1\emptyset INPUT A$
2\emptyset PRINT "A";TAB 5;"B";TAB 1\emptyset;A$
3\emptyset FOR A=\emptyset TO 1
4\emptyset FOR B=\emptyset TO 1
5\emptyset PRINT A;TAB 5;B;TAB 1\emptyset;VAL A$
6\emptyset NEXT B
7\emptyset NEXT A
```

As a response to the INPUT in line 1 \emptyset , you can type in any logical expression involving A and B – for example, A AND B OR B – and the program will print its truth table. Notice the way that the two FOR loops combine to produce all of the possible combinations of values for A and B. You can easily extend this program to work out the truth table for any number of values.

The logical operations AND, OR and NOT can be used on arithmetic expressions. For example, if the variable K contains the value \emptyset , NOT K is one and so on. What is not obvious is the result of

operating on values other than \emptyset and 1. ZX BASIC allows AND, OR and NOT to be used with any values and defines the results as:

X AND Y is X if Y is non-zero
 \emptyset if Y is zero

X OR Y is 1 if Y is non-zero
 X if Y is zero

NOT X is \emptyset if X is non-zero
 1 if X is zero

Notice that these definitions are non-standard, indeed they are entirely arbitrary. There is also an oddity in that the roles of X and Y are not the same in AND and OR when anything other than \emptyset and 1 are used. ZX BASIC also extends the use of logical expressions to strings. However, in this case you can only use AND:

X\$ AND Y\$ is X\$ if Y\$ is non-null
 null if Y\$ is null

Notice that this is the roughly the same definition as given above for general arithmetic values but with the null string playing the role of zero. You might be wondering what use these extended definitions might be? The standard way of using them is similar to the single expression for two rates of tax given above. For example:

TAX=(MONEY*4 \emptyset /1 $\emptyset\emptyset$) AND R\$="HIGH" +
 (MONEY*7 \emptyset /1 $\emptyset\emptyset$) AND R\$="LOW"

This has exactly the same effect as the previous expression but now there is no need to multiply a number by a logical expression. Some people find the first version of this expression easier to understand and some prefer the second – it is all a matter of taste. One thing that is certain is that mixed expressions in ZX BASIC can shorten programs but they do make difficult reading!

Inside the ZX81 – PEEK, POKE and USR

It may come as something of a surprise to learn that BASIC includes a number of commands that allow access to the inner workings of the ZX81. The reason why this might come as a shock is that all the BASIC that we have looked at so far has done its best to avoid getting involved with details of how the machine carries out commands.

However, there are some applications where the normal instructions of BASIC are in some way deficient. For example, they might be too slow or fail to take account of some important feature of the machine. To allow the programmer to find a way around such difficulties, most versions of BASIC include instructions that allow you to 'get at' the inside of the machine.

We learned very early on that a variable is a named area of computer memory. However, it is sometimes necessary to side-step the variable method of using computer memory in preference for direct access using PEEK and POKE. PEEK is a function that will return the contents of a memory location and POKE is a command that will alter the contents of a memory location. It is as simple as that except that you need to know how to specify which memory location and what sort of number can be stored in a memory location. The first problem is easily solved because the ZX81, like all computers, numbers all its memory locations sequentially starting from zero. So PEEK(543) will return the contents of the five-hundred-and-forty-third memory location. The second problem is also easily solved once you know that a memory location can store any number between 0 and 255. So POKE 1000,200 will store 200 in memory location 1000. However POKE 1000,600 will give an error message because 600 is greater than 255. In general, to use PEEK and POKE you have to have a knowledge of what is stored where inside your ZX81 and this is often not easy to find out. PEEKing and POKEing are best avoided unless you are absolutely sure that you know what you are doing.

However, there are one of two 'standard' applications of PEEK and POKE that are worth knowing about and also demonstrate the typical way that PEEK and POKE are used. The ZX81 has a *clock* that ticks in fiftieths of a second buried deep inside it. From the programming point of view it looks like two memory locations that continuously change the values stored in them. The two memory locations work together to extend the range of time that can be held in one memory location. The memory location at 16436 counts fiftieths of a second and, as the largest number that can be stored in a memory location is 255, it counts 255 fiftieths of a second and then goes back to zero. You can think of this as a hand on a clock going round every 255 ticks. The second memory location at 16437 counts how many times the first memory location has 'gone round' and so counts in units of 256×50 of a second. You can make use of this information by

using the PEEK function to discover what is in each memory location and converting it to a number that represents a time in seconds. For example,

$(256 * \text{PEEK } 16437 + \text{PEEK } 16436) / 50$

In the same way you can use POKE to set the two locations to any time that you desire. For example, to zero the clock use:

$\text{POKE } 16437, 0; \text{POKE } 16436, 0$

Most applications of PEEK and POKE are similar in that they require you to know something about where the ZX81 stores some piece of information that is normally hidden from you. For example, if you know that location 16438 is used to store the x co-ordinate of the last point plotted, then you can use PEEK 16438 to discover where you are on the screen.

A function that is often used along with PEEK and POKE is USR. The USR function transfers control out of BASIC and into a 'machine code' program stored somewhere inside the ZX81. Unless you want to get involved in another computer language altogether the USR function will be of little interest to you. However, after you have mastered BASIC the chances are that sooner or later machine code will interest you because only by using machine code can you get the full speed from your ZX81, or any other computer for that matter.

REMark and good programming

The BASIC command REM is the simplest of all in that it does absolutely nothing! Its only purpose is to allow you to include comments that are not part of a program. For example, you could include in every program that you write a first line that tells you what the program is called:

$10 \text{ REM THIS IS MY PROGRAM}$

and the REM would alert the ZX81 to the fact that what followed wasn't to be taken as a line of BASIC but as a note to any humans that might read the program.

You might think it strange that such a simple command is left to the last chapter of a book on BASIC. The reason for this is that although REM is a simple command it can be used in some very sophisticated ways. After you have got over the initial difficulty of writing

programs in BASIC you should look toward trying to write better programs. At first a program that works is a reward in itself but later on a well-written program is what you should aim for. What constitutes a well-written program is something that you will discover for yourself as you learn programming by trial and error and by reading other people's efforts.

The REM statement is part of better programming in that while it certainly isn't necessary it does help to make your programs easier to understand if you include REMs that explain what is happening in each section of your program. Good explanations will help you to return to your programs and re-understand them quickly.

Running out of space - and CLEAR

Although most of the programs in this book will run on a 1K ZX81, programs for *real* applications will almost certainly need more than 1K unless some special tricks are used. There are two attitudes to the sort of tricks needed to squeeze a program into a small amount of RAM. You could treat the problem as part of the fun of computing or you could think of it as an annoying limitation of the machine that is getting in the way of the real computing! If you enjoy taking on difficult challenges for their own sake then you will find lots of fun in squashing programs into 1K of RAM. However, if you are just beginning to write your own programs then there are real problems enough to be found in producing ones that work well without getting involved in tricks. If you enjoy programming our advice is to find more memory rather than restrict your programs - in the future RAM will be cheaper than software!

This said, there is one ZX BASIC command worth knowing about that can be used to make more memory available for a program. The command CLEAR will delete all the variables that a program has used to date and the memory that they used is made available for further use. So, if you were in the middle of a program and found that you needed memory space for an array and you no longer needed any of the variables that you had created you could use CLEAR to delete them. The trouble is, of course, that it is very rare that in the middle of a program you decide that all the variables that you have used up to this point are no longer needed - after all, that program must have been doing some useful work or you wouldn't have bothered running it! In practice the only real use that CLEAR has is

in making room while entering a program. When a program comes to an end because of a STOP or simply because the last line has been obeyed all the variables used still exist. They are only destroyed by the next RUN command. If you find this difficult to believe RUN the following:

```
10 LET A=100
```

and then type in (without a line number) PRINT A. You will find that not only does A still exist after the program has finished but it also contains the value 100. No matter what you do, the variable A will carry on existing until the next time you type RUN when all the variables are removed – unless, that is, you type CLEAR which removes all the variables but doesn't affect the program in any way. The reason why you might want to use CLEAR to remove all the variables before you next RUN the program is fairly obvious. If you are in the middle of adding lines a program that you have just RUN it is possible for you to find that there is insufficient memory to enter a line. If this situation arises all you have to do is type CLEAR and you have all the memory that the variables once used free for editing, etc. In general, however, if you are that short of memory you will probably meet trouble when you RUN the program. Nevertheless, CLEAR can sometimes be useful in desperate situations!

Finding the bugs

There are many reasons why a program might fail. Indeed, it is very rare to write a program that works first time and having mistakes in a program is nothing to be ashamed of. However, the ability to find the bugs within a program is something worth developing. Although it is true that there is no real substitute for experience – the more programs you write and debug the better you will become at it – there is a lot that can be learned just by thinking about how to find errors and faults within a program.

There are two very different types of bug within a program. The first are usually called *syntax errors*, which are things like not matching brackets in an expression or not putting in a comma where one is needed. Syntax errors are rather like spelling mistakes in English and they can usually be found by simply looking at the statement that caused the error. Indeed, the ZX81 itself will alert you to many of the syntax errors you make as it won't let you enter a line

in which it detects such an error. In this respect it is much more helpful than most other computers.

The second type of error is known as a *run time error* because, unlike syntax error, a run time error can often only be found when the program is RUN. For example, a run time error might occur if you try to plot a point that is off the screen. It might not be possible to work out that the point will be off the screen by simply looking at the program, and the error only occurs when the program is RUN. Run time errors are, in general, much more difficult to put right. The reason for this is that the cause of a run time error may not be in the line that actually causes the error. In the case of plotting a point off the screen, the cause of the error might lie in a statement that calculates the co-ordinates of the point to be plotted, not the statement that does the plotting and brings the program to a halt with an error code. Some run time errors can be found by simply staring at the program and checking that it means what you intended it to. What, however, should you do if you still cannot see the error after looking at it for hours?

The first piece of advice is not to stare at the program for hours! Debugging is an 'activity' like programming; you can only achieve a limited amount by just looking. To find a bug all you have to do is check that your program does what you would expect it to do at each point. You must check that the order in which the BASIC statements are carried out is what you would expect it to be and you must check that the variables have the values you would expect. This sounds easy but in a large program it can be a very long job. However, if you follow this advice as much as possible, by trying to match your predictions against what you find in the program, you will at least be gathering information about the bug and actively searching for it rather than just sitting staring at the screen!

Fortunately, ZX BASIC, in common with other versions of BASIC, makes this sort of active debugging particularly easy. The first part of the debugging activity is to check that the program statements are carried out in the order that you would expect. You can do this quite simply by putting STOP statements in at various points of your program that you want to check. If your program reaches one of your STOP commands the program will stop with a code that includes the line number that it reached. To continue execution of your program as if nothing had happened, simply type CONT. Your program will carry on and you will have a valuable

piece of information about the way your program is working. While the program is halted by a STOP command you can also take the opportunity to find out the values stored in the program's variables. In the last section it was explained that all the variables remain preserved until either a RUN or CLEAR is encountered. So, you can type in commands without a line number to print out what each variable contains. The use of the STOP and CONT command coupled with printing out the values stored in variables should be enough to compare what you would expect to find with reality and so pin down the source of the bug.

It has to be admitted that describing how to find a bug is often a lot easier than actually finding one! However, as long as you remember that debugging should be an *activity* and avoid long hours of staring at listings you will find that improvement comes with practice.

Where next?

By now you should have reached a reasonable level of competence in BASIC. But as with most new languages you may feel that, although you can understand and read it, you are less confident about actually writing it. There is only one way to gain confidence and that is to practice and experiment. Certainly, books can help you but only if you are prepared to experiment on your own behalf. Don't be worried if your first programs don't attempt anything very ambitious. It is better to try out your ideas in short and simple routines at first. If you try anything too complicated there is a much greater chance that you'll make mistakes that you can't locate. Try writing program snippets that do just a few things at a time – if you look back through this book you'll find lots of such examples. When all your mini-programs work then it is time to start putting them together to build more extensive ones. The main thing is to go ahead and put what you now know in theory into practice.

Index

AND, 110
arithmetic expression, 28
arithmetic operators, 29
arrays, 65
AT, 85
brackets, 30
cassette recorder, 16
circle, 99
CLEAR, 118
clock, 116
CLS, 88
comma, 85
conditional expressions, 45
constants, 31
CONT, 44
control keys, 9
COPY, 22
debugging, 120
deferred mode, 14
DIM, 66
editing, 15
false, 45
FAST, 103
FOR, 51
functions, 11, 70
GOSUB, 81
GOTO, 41, 47
graphs, 101
high-resolution graphics, 94, 97
IF, 43, 47, 55
immediate mode, 14
index variable, 51
infinite loop, 41
INPUT, 31, 58
iteration, 42
keywords, 11
LET, 27, 58
line, 98
line numbers, 12
LIST, 14, 20
LOAD, 17
loop, 41
low-resolution graphics, 84, 95, 108
NEW, 19
NEWLINE, 11, 13
NEXT, 51
NOT, 110
null string, 62
OR, 110
order of evaluation, 29
PAUSE, 105
PEEK, 115
POKE, 115
PLOT, 94, 97
PRINT, 27, 58, 84
print zone, 85
printer, 22
radians, 75
RAM pack, 23
random numbers, 77
relations, 45
REM, 117
RETURN, 81
reverse video, 14
RUBOUT, 21
RUN, 14, 20
run time errors, 120
SAVE, 17
SCROLL, 88
semicolon, 84
shift key, 9
SLOW, 103
STEP, 53
STOP, 56
strings, 34, 58
string functions, 65, 75
string slicing, 60
subroutines, 80
substrings, 60
syntax errors, 119
TAB, 85
true, 45
truth table, 111
unary minus, 29
UNPLOT, 97
until loop, 50
USR, 117
variable, 25
while loop, 50

THE ZX81

The ZX81 has now become so inexpensive that it puts computer literacy - essential for success in the modern world - within the reach of everybody. It is the ideal microcomputer with which to learn about computers and programming.

Easy to use, with standard facilities found on much more expensive machines, the ZX81 combines fun with tremendous potential for education.

THE ZX81

will teach you BASIC - the world's most popular programming language - in a straightforward and relaxed way. You will start from simple commands and progress to sophisticated techniques, including the use of the ZX81's graphics facilities.

Can you afford to be left behind in the microchip revolution? Find out about computers and programming now!

IN THE SAME SERIES FROM PANTHER BOOKS

THE ZX SPECTRUM
THE DRAGON 32

Front cover photograph by
John Knights

HANDBOOK

U.K. £2.95 NEW ZEALAND \$9.95
AUSTRALIA \$9.95 (recommended)

ISBN 0-586-06105-3



9 780586 061053

THE 2017 GE & JAMES GRANDAD